

Lời nói đầu

Tài liệu này được soạn thảo dùng để hỗ trợ giảng dạy cho môn Cấu trúc dữ liệu và giải thuật trong câu lạc bộ [UPC- UTT Programming Club](#), tại Đại học Công nghệ Giao Thông Vận Tải – Hà Nội. Vì vậy, giáo trình theo sát nội dung cuốn sách Data Structures and Algorithms in Java, Fourth Edition (Adam Drozdek). Tuy nhiên, để đảm bảo chất lượng giáo trình, mình đã tham khảo nhiều sách khác về cấu trúc dữ liệu và giải thuật, cố gắng trình bày ngắn gọn và cô đọng các kiến thức cần nắm vững.

Mình là Hoàng Anh Tiến một thành viên của [UPC- UTT Programming Club](#) đặc biệt cảm ơn tới câu lạc bộ đã cho mình động lực viết giáo trình này.

Mặc dù đã dành rất nhiều thời gian và công sức cho tài liệu này, phải hiệu chỉnh chi tiết và nhiều lần, nhưng tài liệu không thể nào tránh được những sai sót và hạn chế. Mình thật sự mong nhận được các ý kiến góp ý từ bạn đọc để tài liệu có thể hoàn thiện hơn.

Các bạn trong câu lạc bộ nếu có sử dụng giáo trình này, xin gửi cho mình ý kiến đóng góp phản hồi, giúp giáo trình được hoàn thiện thêm, phục vụ cho công tác giảng dạy chung và góp phần giúp câu lạc bộ phát triển và lớn mạnh hơn.

Phiên bản

Cập nhật ngày: [02/09/2024](#)

Thông tin liên lạc

Mọi ý kiến và câu hỏi có liên quan xin vui lòng gửi về:

- + Địa chỉ: 54, Triều Khúc, P. Thanh Xuân Nam, Q. Thanh Xuân, Hà Nội
- + Sđt: 033 228 4267
- + Fanpage: [UPC- UTT Programming Club](#)
- + Email: uttprogrammingclub@gmail.com

Phân tích giải thuật

Cùng một vấn đề có thể giải quyết bằng các giải thuật với hiệu quả khác nhau. Các giải thuật không khác biệt nhiều khi xử lý với số lượng dữ liệu nhỏ, nhưng khác biệt sẽ tăng đáng kể với số lượng dữ liệu lớn. Ta cần kỹ thuật dùng để so sánh hiệu quả của các giải thuật mà không cần cài đặt cụ thể chúng, Juris Hartmanis và Richard Stearns đã phát triển khái niệm tính toán độ phức tạp (computational complexity) để giải quyết vấn đề này.

Tính toán độ phức tạp cho thấy *chi phí* cần cho việc áp dụng giải thuật, chủ yếu là *không gian lưu trữ và thời gian thực hiện*, trong đó yếu tố thời gian thực hiện quan trọng hơn. Yếu tố này được xem xét tách biệt với việc cài đặt trên một hệ thống máy tính cụ thể. Nếu ước lượng được số lượng các phép toán cần thực hiện của giải thuật và biết được thời gian cần thực hiện một phép toán cơ bản trên một hệ thống máy tính cụ thể, thì cũng có nghĩa là ta ước lượng được thời gian thực hiện giải thuật đó.

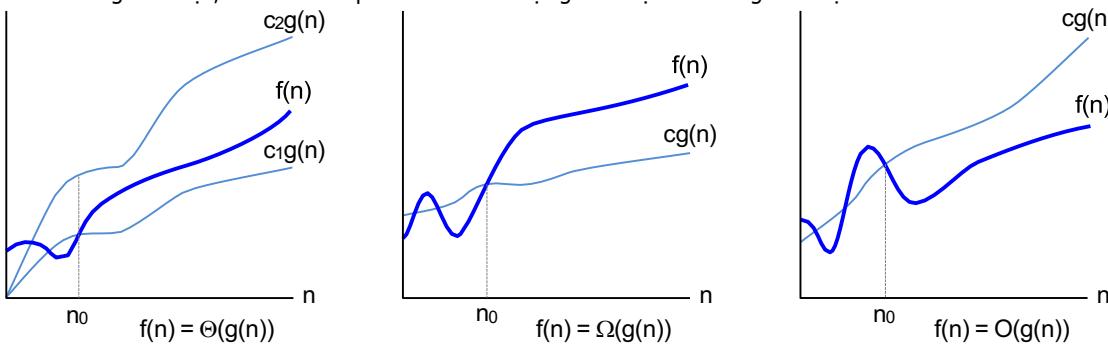
I. Đánh giá độ phức tạp

Giải thuật được phân lớp bởi các hàm đánh giá độ phức tạp của nó. Các hàm này mô tả mối liên quan giữa thời gian thực hiện của giải thuật với kích thước của bài toán (số lượng dữ liệu cần xử lý) mà giải thuật giải quyết. Nói cách khác, thời gian thực hiện giải thuật $f(n)$ là một hàm theo n , kích thước dữ liệu đầu vào cần xử lý.

1. Ký pháp (notation) đánh giá độ phức tạp

Cho hai hàm $f(n)$ và $g(n)$ là các hàm định trị dương với mọi n , $f(n)$ là thời gian thực hiện một giải thuật. Ta nói độ phức tạp của giải thuật có thời gian thực hiện $f(n)$ là:

- Theta-lớn $\Theta(g(n))$ nếu tồn tại các số dương c_1, c_2 và n_0 sao cho $c_1g(n) \leq f(n) \leq c_2g(n)$, với mọi $n \geq n_0$. Hàm $g(n)$ được gọi là *giới hạn chặt* (asymptotically tight bound) của hàm $f(n)$ với n đủ lớn. Nếu ta đánh giá được độ phức tạp của giải thuật bằng Θ -lớn, có thể coi kết quả đánh giá là đủ chặt và không cần đánh giá bằng những ký pháp khác.
- Omega-lớn $\Omega(g(n))$ nếu tồn tại các số dương c và n_0 sao cho $f(n) \geq cg(n)$, với mọi $n \geq n_0$. Hàm $g(n)$ được gọi là *giới hạn dưới* (asymptotic lower bound) của hàm $f(n)$ với n đủ lớn. Omega-lớn thường dùng khi đề cập đến chi phí tối thiểu của giải thuật.
- O-lớn $O(g(n))$ nếu tồn tại các số dương c và n_0 sao cho $f(n) \leq cg(n)$, với mọi $n \geq n_0$. Hàm $g(n)$ được gọi là *giới hạn trên* (asymptotic upper bound) của hàm $f(n)$ với n đủ lớn. O-lớn được giới thiệu bởi Paul Bachmann, gọi là Big-O. Thường dùng để đề cập đến tính tốt của giải thuật, so sánh chi phí tối đa của một giải thuật với các giải thuật khác.



Minh họa các ký pháp Θ , Ω và O , $f(n)$ là thời gian thực hiện giải thuật, với n đủ lớn ($n \geq n_0$).

Khi đánh giá độ phức tạp của giải thuật bằng Big-O, thay vì dựa trên thời gian thực hiện $f(n)$, ta dựa trên $g(n)$, gọi là *tỷ suất tăng* (growth rate) của $f(n)$.

Xem ví dụ sau, ta có hai giải thuật P_1 và P_2 với thời gian thực hiện tương ứng là $f_1(n) = 80n^2$ (có tỷ suất tăng là n^2) và $f_2(n) = 4n^3$ (có tỷ suất tăng là n^3). Với $n < 20$, P_2 sẽ tăng nhanh hơn P_1 ($f_2(n) < f_1(n)$) vì hệ số của $4n^3$ nhỏ hơn hệ số của $80n^2$, nhưng với $n > 20$, P_1 sẽ tăng nhanh hơn P_2 do số mũ của $4n^3$ lớn hơn số mũ của $80n^2$.

Ta thường chỉ quan tâm đến trường hợp n đủ lớn, $n \geq n_0$, trong ví dụ trên $n_0 = 20$. Vì với n nhỏ thời gian thực hiện của P_1 và P_2 đều không lớn và sự khác biệt giữa $f_1(n)$ và $f_2(n)$ là không đáng kể. Nói cách khác, với n đủ lớn, $g(n)$ tăng nhanh hơn $f(n)$. Vì vậy, đánh giá giải thuật dựa vào tỷ suất tăng $g(n)$ hợp lý hơn dựa vào hàm thời gian $f(n)$.

2. Phân lớp các giải thuật

Ta gọi giải thuật có $f(n)$ là $O(g(n))$, hoặc giải thuật đang xét thuộc lớp giải thuật có độ phức tạp $O(g(n))$. Như vậy, ký pháp Big-O nhóm các giải thuật vào các lớp có $g(n)$ khác nhau. Nói chung có hai lớp chính:

- Lớp đa thức: $O(1), O(n), O(n^2), O(n^3)$. Các giải thuật $O(\lg n), O(n \lg n)$ cũng liệt kê trong lớp đa thức vì bị chặn trên bởi n^2 . Các giải thuật thuộc lớp đa thức được xem là có chi phí chấp nhận được, có thể áp dụng trong thực tế.
- Lớp không đa thức: lớp lũy thừa $O(2^n)$, lớp giai thừa $O(n!)$. Các giải thuật thuộc lớp này chỉ áp dụng với n nhỏ, cần phải cải tiến giải thuật để giảm độ phức tạp.

Các hàm $g(n)$ được sắp xếp theo tốc độ tăng: k (hàm hằng), $\lg n$, n , $n \lg n$, n^2 , n^3 , 2^n , $n!$. Mỗi hàm đứng sau là Big-O của các hàm đứng trước nó, gọi là quy tắc ưu thế (dominant).

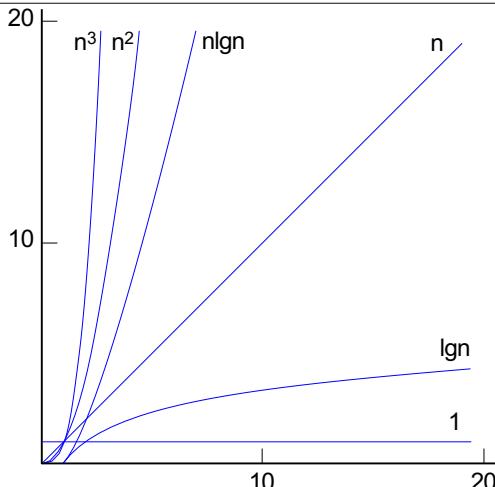
Ví dụ: $f(n) = 7n^2 + 3n \lg n + 5n + 1000$ là $O(n^2)$.

Bạn cần phân biệt *lớp các giải thuật* với *lớp các bài toán* (problem):

- Lớp P: lớp các bài toán có thể giải được trong thời gian đa thức.
 - Lớp NP: lớp các bài toán có thể kiểm tra lời giải của chúng trong thời gian đa thức.
 - Lớp NP-hard: lớp các bài toán không có giải thuật trong thời gian đa thức để giải nó, ngoại trừ $P = NP$.
- Nói cách khác, một bài toán thuộc lớp NP-hard nếu tồn tại giải thuật giải nó trong thời gian đa thức, kéo theo sự tồn tại giải thuật đa thức giải mọi bài toán trong NP. Nghĩa là, mọi bài toán trong NP có thể quy về bài toán đó trong thời gian đa thức.
- Lớp NP-completeness: lớp các bài toán thuộc NP và là NP-hard.

Bảng sau là các lớp giải thuật và thời gian thực hiện chúng trên hệ thống máy tính có thể thực hiện một triệu tác vụ/giây.

Lớp giải thuật	Độ phức tạp	Số tác vụ và thời gian thực hiện (1 chỉ thị/μsec)					
n		10	10 ²	10 ³	10 ⁴	10 ⁵	10 ⁶
const (hằng)	O(1)	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	O(lgn)	3.32	3 μsec	6.64	7 μsec	9.97	10 μsec
linear (tuyến tính)	O(n)	10	10 μsec	10 ²	100 μsec	10 ³	1 msec
superlinear	O(nlgn)	33.2	33 μsec	664	664 μsec	9970	10 msec
quadratic (bậc hai)	O(n ²)	10 ²	100 μsec	10 ⁴	10 msec	10 ⁶	1 sec
cubic (bậc ba)	O(n ³)	10 ³	1 msec	10 ⁶	1 sec	10 ⁹	16.7 min
exponential (lũy thừa)	O(2 ⁿ)	1024	10 msec	10 ³⁰	3.17x10 ¹⁷ năm	10 ³⁰¹	
n		10 ⁴	10 ⁵	10 ⁶			
const (hằng)	O(1)	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	O(lgn)	13.3	13 μsec	16.6	17 μsec	19.93	20 μsec
linear (tuyến tính)	O(n)	10 ⁴	10 msec	10 ⁵	0.1 sec	10 ⁶	1 sec
superlinear	O(nlgn)	133x10 ³	133 msec	166x10 ⁴	1.6 sec	199.3x10 ⁵	20 sec
quadratic (bậc hai)	O(n ²)	10 ⁸	1.7 min	10 ¹⁰	16.7 min	10 ¹²	11.6 ngày
cubic (bậc ba)	O(n ³)	10 ¹²	11.6 ngày	10 ¹⁵	31.7 năm	10 ¹⁸	31709 năm
exponential (lũy thừa)	O(2 ⁿ)	10 ³⁰¹⁰	10 ³⁰¹⁰³			10 ³⁰¹⁰³⁰	



Phân lớp giải thuật bằng Big-O, các hàm g(n).

II. Các thuộc tính của Big-O

Ký pháp Big-O có một số thuộc tính hữu dụng cho việc ước lượng tính hiệu quả của giải thuật.

- Tính bắc cầu (transitivity): nếu $f(n)$ là $O(g(n))$ và $g(n)$ là $O(h(n))$ thì $f(n)$ là $O(h(n))$. Nói cách khác: $O(O(g(n)))$ là $O(g(n))$.
- Nếu $f(n)$ là $O(h(n))$ và $g(n)$ là $O(h(n))$ thì $f(n) + g(n)$ là $O(h(n))$.
- $f(n) = a n^k$ là $O(n^k)$.
- $f(n) = n^k$ là $O(n^{k+j})$ với mọi j là số dương.
- Nếu $f(n) = cg(n)$ thì $f(n)$ là $O(g(n))$.
- $f(n) = \log_a n$ là $O(\log_b n)$ với mọi cơ số dương a và $b \neq 1$.
- $f(n) = \log_a n$ là $O(\lg n)$ với mọi cơ số dương $a \neq 1$, ở đây $\lg n = \log_2 n$.

Ngoài ra, có một số quy tắc giúp xác định độ phức tạp của giải thuật:

- Quy tắc bỏ hằng số: nếu $f(n)$ là $O(cg(n))$ với c là hằng số dương, thì $f(n)$ là $O(g(n))$.

Từ đó, $f(k)$ là $O(1)$, với k là hằng số dương.

- Quy tắc lấy max: nếu $f(n)$ là $O(g(n) + h(n))$, thì $f(n)$ là $O(\max(g(n), h(n)))$.

Từ đó, đa thức có O (đa thức có lũy thừa cao nhất). Ví dụ: $f(n) = 7n^4 + 3n^2 + 1000$ là $O(n^4)$

- Quy tắc cộng: nếu $f_1(n)$ là $O(g(n))$ và $f_2(n)$ là $O(h(n))$, thì $f_1(n) + f_2(n)$ là $O(g(n) + h(n))$. Thường dùng xác định thời gian thực hiện một chuỗi lệnh tuần tự.

- Quy tắc nhân: nếu $f_1(n)$ là $O(g(n))$ và $f_2(n)$ là $O(h(n))$, thì $f_1(n)f_2(n)$ là $O(g(n)h(n))$.

III. Trường hợp tốt nhất, xấu nhất và trung bình

Nhiều trường hợp thời gian thực hiện không chỉ phụ thuộc vào kích thước của bài toán, mà còn phụ thuộc vào nội dung, đặc tính, tình trạng thực tế của dữ liệu đầu vào. Điều hình là bài toán sắp xếp mảng các phần tử, cùng số phần tử nhưng sắp xếp mảng các phần tử nghịch thế mất nhiều thời gian hơn sắp xếp mảng đã đúng thứ tự. Khi đó, ngoài việc xác định thời gian thực hiện giải thuật, ta còn phải xem xét trường hợp tốt nhất, trường hợp xấu nhất và trường hợp trung bình.

Ví dụ: phương thức `isMember()` dùng giải thuật so sánh tuyến tính để xác định target có phải là thành viên của mảng data.

```
boolean isMember(E[] data, E target) {
    for (int i = 0; i < data.length; ++i)
        if (data[i].compareTo(target) == 0)
            return true;
    return false;
}
```

Ta chú ý đến phép so sánh trong if, gọi là *phép toán tích cực*. Phép toán tích cực là phép toán có số lần thực hiện không ít hơn các phép toán khác trong giải thuật. Số lần thực hiện phép toán tích cực giúp ta ước lượng được số thao tác cần thực hiện.

Trường hợp tốt nhất (best-case), ta mất thời gian ít nhất khi thực hiện giải thuật, xảy ra khi phần tử đầu tiên trong mảng là target. Thời gian thực hiện trong trường hợp tốt nhất là $\Theta(1)$. Phân tích trường hợp tốt nhất không có nhiều hữu ích, một giải thuật có thể rất tốt trong một số trường hợp hiếm xảy ra, nhưng lại có hiệu suất thấp nói chung.

Trường hợp xấu nhất (worst-case), ta mất thời gian lớn nhất khi thực hiện giải thuật, xảy ra khi target không có trong mảng. Dẫn đến thời gian thực hiện là $O(n)$.

Phân tích trường hợp trung bình (average-case) là khó nhất, nó yêu cầu một số giả định thế nào là tình trạng "trung bình" của dữ liệu. Cho một tập các sự kiện (events) khác nhau có thể xảy ra, thời gian thực hiện trung bình là:

$$\sum_{i \in events} P(i) t(i)$$

Trong đó, $P(i)$ là xác suất xuất hiện sự kiện i , $t(i)$ thời gian thực hiện nếu sự kiện i xuất hiện.

Việc chọn các sự kiện phải bảo đảm là toàn diện (exhaustive, ít nhất một trong chúng sẽ xảy ra) và loại trừ lẫn nhau (mutually exclusive, một thời điểm chỉ một sự kiện xảy ra). Ta giả sử target chỉ có một và khả năng chọn nó là giống như chọn các phần tử khác. Như vậy, xác suất xuất hiện của target là $1/n$.

Nếu target có chỉ số 0, ta mất 1 bước trong vòng lặp; nếu target có chỉ số i , ta mất $(i + 1)$ bước trong vòng lặp để tìm ra nó.

Thời gian thực hiện trung bình là:

$$\sum_{i=0}^{n-1} \frac{1}{n} (i + 1) = \frac{1}{n} \sum_{i=1}^n i = \frac{n+1}{2} \in O(n)$$

Nghĩa là, trong trường hợp trung bình, thời gian thực hiện của giải thuật gần với trường hợp xấu nhất, và không bao giờ tốt hơn trường hợp tốt nhất.

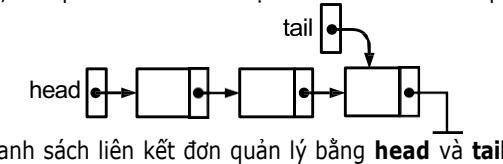
Danh sách liên kết

Mảng (array) là cấu trúc dữ liệu sử dụng phổ biến trong nhiều ngôn ngữ lập trình. Tuy nhiên, mảng có một số khuyết điểm:

- Phải biết trước kích thước của mảng tại thời điểm biên dịch.
- Các phần tử của mảng được lưu trữ kế tiếp nhau. Vì vậy, khi xóa (hoặc chèn) tại một vị trí phải dồn các phần tử của mảng từ sau vị trí đó về phía trước (hoặc ra phía sau). Độ phức tạp của thao tác chèn và xóa là O(n).
- Vùng nhớ cấp phát cho mảng không được thu hồi kể cả khi chúng không còn cần thiết.

Danh sách liên kết, với khả năng cấp phát động, giải quyết được các vấn đề trên.

Danh sách liên kết là tập hợp các phần tử, mỗi phần tử chứa dữ liệu và liên kết chỉ đến phần tử khác.



I. Danh sách liên kết đơn (Singly Linked List)

Mỗi node, ngoài dữ liệu còn có liên kết **next** chỉ đến node kế tiếp.

```
public class SLLNode<E> {
    protected E info;
    protected SLLNode<E> next;

    public SLLNode(E info, SLLNode<E> next) {
        this.info = info;
        this.next = next;
    }

    public SLLNode(E info) {
        this(info, null);
    }
}
```

Chỉ cần một liên kết chỉ đến node đầu của danh sách liên kết đơn, gọi là **head**, để quản lý danh sách liên kết đơn. Tuy nhiên, để chèn/xóa nhanh node cuối, ta cần thêm liên kết **tail** chỉ đến node cuối của danh sách liên kết đơn.

```
public class SLL<E extends Comparable<? super E>> {
    protected SLLNode<E> head = null;
    protected SLLNode<E> tail = null;
}
```

Để nắm vững việc thao tác trên danh sách liên kết, ta cần xác định chính xác những *thông tin phải cập nhật* khi thay đổi danh sách. Ngoài ra, bạn phải luôn luôn chú ý theo dõi cập nhật **head** và **tail** nếu chúng bị thay đổi.

1. Chèn thêm node

a) Chèn đầu

Độ phức tạp O(1).

Nếu danh sách đang rỗng, node mới chèn vào có **next** chỉ đến **null** vì nó cũng là node cuối. Chèn node mới làm thay đổi cả **tail**, cập nhật **tail** chỉ đến node mới. Cần cập nhật:

- **next** của node mới, chỉ đến **null**.
- **head** và **tail** chỉ đến node mới.

Nếu danh sách không rỗng, node mới chèn vào có **next** chỉ đến node đầu (hiện do **head** chỉ đến). Chèn node mới chỉ ảnh hưởng đến **head**. Cần cập nhật:

- **next** của node mới, chỉ đến node đầu.
- **head**, chỉ đến node mới.

```
public void addToHead(E info) {
    if (isEmpty()) {
        head = tail = new SLLNode<>(info);
    } else {
        head = new SLLNode<>(info, head);
    }
}
```

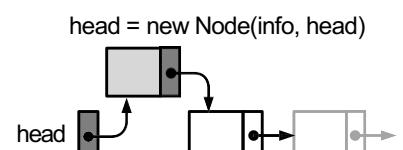
```
public boolean isEmpty() {
    return (head == null);
}
```

b) Chèn cuối

Tác vụ này có sự tham gia của liên kết **tail**, độ phức tạp O(1). Nếu không có **tail**, do ta phải di chuyển đến cuối danh sách để chèn, độ phức tạp O(n).

Nếu danh sách rỗng, tương tự chèn đầu khi danh sách rỗng. Cần cập nhật:

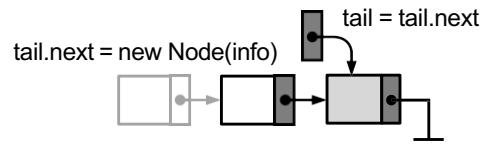
- **next** của node mới, chỉ đến **null**.
- **head** và **tail** chỉ đến node mới.



Nếu danh sách không rỗng, node mới chèn vào là node cuối nên **next** của nó chỉ đến **null**. Chèn node mới chỉ ảnh hưởng đến **tail**. Cần cập nhật:

- **tail.next**, chính là **next** của node cuối cũ, trước đây chỉ **null**, bây giờ chỉ đến node mới.
- **tail** chỉ đến node mới.

```
public void addToTail(E info) {
    if (isEmpty()) {
        head = tail = new SLLNode<>(info);
    } else {
        tail.next = new SLLNode<>(info);
        tail = tail.next;
    }
}
```



c) Chèn tại vị trí nào đó theo yêu cầu

Độ phức tạp O(n).

Cần có tham chiếu chỉ đến node *ngay trước* vị trí cần chèn, để có thể thay đổi **next** của nó chỉ đến node mới. Nếu chèn đầu thì không có "node ngay trước" này, nên ta cần phải xét riêng trường hợp chèn đầu.

Một phương pháp thú vị là dùng node giả (ghost node) gắn thêm vào đầu danh sách, giải thuật sẽ đơn giản hơn vì ta không cần xét trường hợp chèn đầu, chèn đầu nghĩa là chèn sau node giả.

Thủ tục sau sẽ chèn một node vào danh sách tăng sao cho danh sách vẫn giữ đúng tính chất tăng. Để có tham chiếu đến "node ngay trước" ta dùng thủ thuật "dò trước": di chuyển với **t** và kiểm tra bằng **t.next**. Cụ thể, di chuyển **t** tới trước cho đến khi kiểm tra thấy **t.next** chỉ đến vị trí cần chèn thì **t** là "node ngay trước" vị trí cần chèn.

```
public void insertAscending(E info) {
    SLLNode<E> t, p = new SLLNode<E>(null, head); // thêm node giả (ghost node, p)
    for (t = p; t.next != null && t.next.info.compareTo(info) < 0; t = t.next) { }
    t.next = new SLLNode<E>(info, t.next);
    if (tail == t) tail = t.next; // cập nhật tail nếu chèn cuối
    head = p.next; // loại node giả, cập nhật lại head nếu chèn đầu
}
```

2. Xóa node

a) Xóa đầu

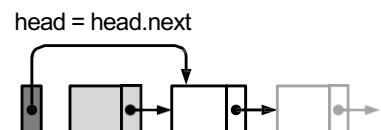
Độ phức tạp O(1).

- dữ liệu node đầu được lưu tạm lại để trả về sau khi xóa.

- cập nhật **head** chỉ đến node kế tiếp (**head.next**), nghĩa là **head** "tiến lên" bỏ lại node đầu.

Nếu danh sách chỉ có một node, xóa node đó làm danh sách rỗng nên thay đổi cả **tail**, vì vậy cập nhật **tail** thành **null**.

```
public E deleteFromHead() {
    E data = null;
    if (!isEmpty()) {
        data = head.info;
        if (head == tail) head = tail = null; // danh sách chỉ có một node
        else head = head.next;
    }
    return data;
}
```



b) Xóa cuối

Độ phức tạp O(n).

- dữ liệu node đầu được lưu tạm lại để trả về sau khi xóa.

Để cập nhật **next** của node trước **tail** nên ta cần biết node trước **tail**, gọi là **p**.

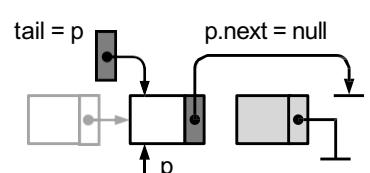
Danh sách liên kết đơn không đi ngược lên được, từ **tail** không xác định được **p** trước nó, ta phải tìm **p** bằng vòng lặp chạy từ đầu danh sách. Ta dùng thủ thuật "dò trước": chạy cho đến khi kiểm tra thấy **p.next** chỉ đến **tail** thì **p** là node trước **tail** (node kế cuối).

Sau đó cập nhật:

- **p.next** thành **null**, dẫn đến loại bỏ node cuối.
- **tail** chỉ đến **p**.

Nếu danh sách chỉ có một node, xóa node đó làm danh sách rỗng nên thay đổi cả **head**, cập nhật **head** thành **null**.

```
public E deleteFromTail() {
    E data = null;
    if (!isEmpty()) {
        data = tail.info;
        if (head == tail) head = tail = null;
        else {
            SLLNode<E> p = head;
            for (; p.next != tail; p = p.next) { }
            p.next = null;
            tail = p;
        }
    }
    return data;
}
```



c) Xóa node bất kỳ

Độ phức tạp O(n).

Ta phải xét các trường hợp:

- node cần xóa là node đầu, có hai trường hợp con:

+ danh sách có một node duy nhất và cũng là node cần xóa, cập nhật **head** và **tail** thành **null** để xóa node đó.

+ danh sách có nhiều node và node cần xóa là node đầu, cho **head** "tiến lên" xóa node đầu, không cần cập nhật **tail** do trong trường hợp này danh sách có từ hai node trở lên.

- node cần xóa là node bất kỳ, không phải node đầu, cần tìm node **pred** nằm ngay trước node cần xóa.

Ta tìm node **pred** bằng thủ thuật "hai con trỏ theo nhau": **pred** luôn theo ngay sau **tmp**, **tmp** "tiến lên" thì **pred** cũng "tiến lên"; di chuyển cho đến khi **tmp** chỉ đến ngay node cần xóa, **pred** sẽ chỉ đến node nằm ngay trước **tmp**.

Vòng lặp tìm node cần xóa có thể thất bại khi không tìm ra node cần xóa, khi đó **tmp** chỉ đến **null**.

Nếu **tmp** khác **null**, ta đã tìm được node cần xóa, **pred** là node nằm ngay trước nó, ta cập nhật **pred**:

- **next** của node **pred** chỉ đến sau node cần xóa: **pred.next = tmp.next**

- nếu node cần xóa là **tail**, cập nhật lại **tail**.

```
public E delete(E info) {
    E data = null;
    if (!isEmpty()) {
        if (head == tail && info.compareTo(head.info) == 0) {
            data = head.info;
            head = tail = null;
        } else if (info.compareTo(head.info) == 0) {
            data = head.info;
            head = head.next;
        } else {
            SLLNode<E> pred, tmp;
            for (pred = head, tmp = head.next;
                 tmp != null && tmp.info != info;
                 pred = pred.next, tmp = tmp.next) { }
            if (tmp != null) {
                data = tmp.info;
                pred.next = tmp.next;
                if (tmp == tail) tail = pred;
            }
        }
    }
    return data;
}
```

Một phương pháp thú vị là quy về trường hợp xóa đầu: chép đè dữ liệu của node đầu lên dữ liệu của node cần xóa, như vậy có hai node chứa dữ liệu giống nhau (node đầu và node cần xóa), xóa node đầu. Ta còn gọi là phương pháp xóa bằng sao chép.

public E delete(E info) {

```
    E data = null;
    SLLNode<E> tmp;
    // vòng lặp tìm node cần xóa
    for (tmp = head; tmp != null && tmp.info.compareTo(info) != 0; tmp = tmp.next) { }
    if (tmp != null) {
        data = tmp.info;
        tmp.info = head.info;
        deleteFromHead();
    }
    return data;
}
```

Danh sách liên kết là cấu trúc dữ liệu thích hợp khi dữ liệu lưu trữ bị thay đổi (thêm vào, loại bỏ) với tần số cao. Tìm kiếm trên danh sách liên kết vẫn tuyến tính giống như mảng, độ phức tạp O(n).

Khi truy cập ngẫu nhiên, mảng có ưu điểm hơn danh sách liên kết do cho phép truy cập bằng chỉ số. Điều này phù hợp khi tìm kiếm nhị phân hay cho hầu hết các giải thuật sắp xếp. Mảng cũng tiết kiệm vùng nhớ hơn vì không có các liên kết.

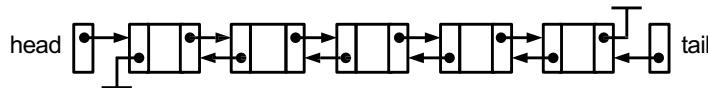
II. Danh sách liên kết đôi (Doubly Linked List)

Với danh sách liên kết đơn một node không giữ tham chiếu đến node ngay trước nó, ta không thể đi ngược từ cuối lên đầu trong danh sách liên kết đơn.

Điều này dẫn đến một số bất tiện, ví dụ khi xóa một node bất kỳ, bao gồm cả xóa node cuối, đều có độ phức tạp O(n) do phải xác định được node nằm ngay trước node cần xóa.

Sử dụng danh sách liên kết đôi có thể giải quyết vấn đề trên.

Trong danh sách liên kết đôi, mỗi node ngoài dữ liệu nó chứa, còn có hai liên kết: **next** chỉ đến node kế tiếp và **prev** chỉ đến node nằm ngay trước nó.



Danh sách liên kết đôi, mỗi node có hai liên kết **next** và **prev**.

```

public class DLLNode<E> {
    protected DLLNode<E> prev;
    protected E info;
    protected DLLNode<E> next;

    public DLLNode(DLLNode<E> prev, E info, DLLNode<E> next) {
        this.prev = prev;
        this.info = info;
        this.next = next;
    }

    public DLLNode(E info) {
        this(null, info, null);
    }
}

```

Vì có thêm một liên kết **prev**, các thao tác trên danh sách liên kết đôi phải chú ý cập nhật liên kết này nếu có thể, dẫn đến cài đặt thêm phức tạp. Tuy nhiên, việc có thêm liên kết **prev** giúp giảm độ phức tạp của các tác vụ.

```

public class DLL<E extends Comparable<? super E>> {
    protected DLLNode<E> head = null;
    protected DLLNode<E> tail = null;
}

```

Xem xét phương thức xóa một node cuối danh sách, có độ phức tạp O(n) trên danh sách liên kết đơn, do phải xác định node trước node cuối. Trên danh sách liên kết đôi, node trước node cuối được tìm ra ngay bằng **tail.prev** nên độ phức tạp của phương thức này giảm thành O(1):

```

public E deleteFromTail() {
    E data = null;
    if (!isEmpty()) {
        data = tail.info;
        if (head == tail) head = tail = null;
        else {
            tail = tail.prev;
            tail.next = null;
        }
    }
    return data;
}

```

```

public boolean isEmpty() {
    return (head == null);
}

```

Ví dụ khác, phương thức xóa một node bất kỳ trong danh sách liên kết đôi. Ta nhận thấy:

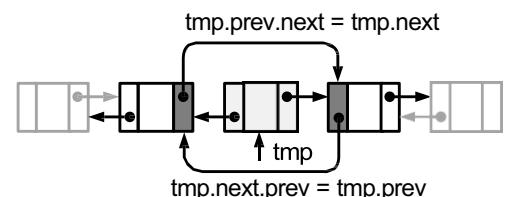
- Khi cập nhật cần chú ý cập nhật thêm liên kết **prev**.
- Độ phức tạp O(1) (không tính chi phí tìm kiếm node xóa) do ta xác định được ngay node nằm ngay trước node cần xóa **tmp**, chính là **tmp.prev**.

```

public E delete(E info) {
    E data = null;
    if (!isEmpty()) {
        if (head == tail && info.compareTo(head.info) == 0) {
            data = head.info;
            head = tail = null;
        } else if (info.compareTo(head.info) == 0) {
            data = head.info;
            head = head.next;
            head.prev = null;
        } else {
            DLLNode<E> tmp;
            for (tmp = head;
                 tmp != null && tmp.info.compareTo(info) != 0;
                 tmp = tmp.next) { }

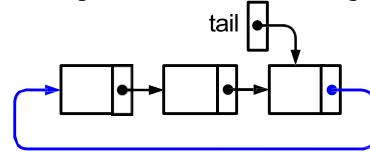
            if (tmp != null) {
                data = tmp.info;
                tmp.prev.next = tmp.next;
                if (tmp == tail) tail = tmp.prev;
                else tmp.next.prev = tmp.prev;
            }
        }
    }
    return data;
}

```



III. Danh sách liên kết vòng (Circular Linked List)

Con trỏ **null** duy nhất trong danh sách liên kết đơn, **tail.next**, được tận dụng để chỉ đến node đầu thay cho **head**; ta có danh sách liên kết vòng. Danh sách liên kết vòng vẫn dùng cấu trúc SLLNode nhưng không có con trỏ **null**.



Danh sách liên kết vòng đơn, **head** là **tail.next**.

```
public class CLL<E extends Comparable<? super E>> {
    protected SLLNode<E> tail = null;
}
```

Do không cần **head** nữa, danh sách liên kết vòng được quản lý chỉ bằng liên kết **tail**. Tham chiếu **head** được ngầm hiểu chính là **tail.next**.

So sánh một số phương thức của danh sách liên kết vòng với phương thức tương tự trên danh sách liên kết đơn, ta nhận thấy ý tưởng **head** là **tail.next** thể hiện rất rõ:

- Chèn đầu: chèn sau **tail.next**:

```
public void addToHead(E info) {
    if (isEmpty()) {
        tail = new SLLNode<>(info);
        tail.next = tail;
    } else {
        tail.next = new SLLNode<>(info, tail.next);
    }
}
```

```
public boolean isEmpty() {
    return (tail == null);
}
```

- Chèn cuối: chèn sau **tail**.

```
public void addToTail(E info) {
    if (isEmpty()) {
        tail = new SLLNode<>(info);
        tail.next = tail;
    } else {
        tail.next = new SLLNode<>(info, tail.next);
        tail = tail.next;
    }
}
```

- Xóa một node bất kỳ: việc không có con trỏ **null** cũng gây trở ngại khi phải xác định duyệt đến cuối danh sách.

```
public E delete(E info) {
    E data = null;
    if (!isEmpty()) {
        if (tail == tail.next && info.compareTo(tail.info) == 0) {
            data = tail.info;
            tail = null;
        } else if (info.compareTo(tail.next.info) == 0) {
            data = tail.next.info;
            tail.next = tail.next.next;
        } else {
            SLLNode<E> pred, tmp;
            for (pred = tail.next, tmp = pred.next;
                 tmp != tail.next && tmp.info.compareTo(info) != 0; // (*)
                 tmp = tmp.next, pred = pred.next) { }
            if (tmp != tail.next) { // (*)
                data = tmp.info;
                pred.next = tmp.next;
                if (tmp == tail) tail = pred;
            }
        }
    }
    return data;
}
```

(*) Vì **tail.next** (tức **head**) đã xét trong các trường hợp trước đó nên ta dùng **tmp != tail.next** để làm điều kiện dừng vòng lặp khi duyệt hết danh sách.

Khi duyệt danh sách liên kết vòng, bạn cũng có thể tạm thời cắt nó thành danh sách liên kết đơn rồi phục hồi trở lại:

```
public void printAll() {
    if (!isEmpty()) {
        SLLNode<E> head = tail.next;
        tail.next = null;
```

```
for (SLLNode<E> p = head; p != null; p = p.next)
    System.out.println(p.info);
tail.next = head;
}
```

Ngoài danh sách liên kết vòng đơn, ta cũng có danh sách liên kết vòng đôi.

Ngăn xếp - Hàng đợi

I. Stack

Stack (ngăn xếp) là một cấu trúc dữ liệu tuyến tính, chỉ được truy cập tại m vị trí gọi là đỉnh stack (top).

Phần tử đưa vào vị trí top sau cùng sẽ được lấy ra đầu tiên, đó là nguyên tắc hoạt động LIFO (Last In First Out) của stack. Các phần tử của stack được lấy ra theo thứ tự ngược lại so với khi các phần tử này được đưa vào stack.

Stack được cài đặt bằng mảng hoặc bằng danh sách liên kết. Nếu stack được cài đặt bằng danh sách liên kết, top chính là head của danh sách liên kết đó.

1. Các thao tác với stack

- **isEmpty()**: tương đương với việc kiểm tra danh sách liên kết có rỗng hay không.
- **push()**: độ phức tạp $O(1)$. Đưa thêm phần tử vào đỉnh stack tương đương với *chèn đầu* vào danh sách liên kết. Nếu cài đặt bằng mảng, chú ý ném exception khi đưa một phần tử vào stack đã đầy, gây lỗi *stack overflow*. Thay vì ném exception, bạn có thể cấp phát mới và chuyển dữ liệu từ mảng cũ sang mảng mới, khi đó độ phức tạp là $O(n)$.
- **pop()**: độ phức tạp $O(1)$. Lấy phần tử ra khỏi đỉnh stack tương đương với *xóa đầu* danh sách liên kết. Chú ý ném exception khi lấy phần tử ra khỏi một stack đang rỗng, gây lỗi *stack underflow*.
- **topEl()**: còn gọi là peek, trả về phần tử tại đỉnh stack mà không cần lấy nó ra khỏi stack. Chú ý ném exception khi peek một stack đang rỗng.
- **clear()**: xóa toàn bộ stack, làm rỗng stack.

Nếu cài đặt stack bằng mảng, khi phải thay đổi kích thước của mảng, thường theo các nguyên tắc sau:

+ *repeated doubling*: trong khi push() nếu mảng đầy, ta cấp phát mảng mới có *kích thước gấp đôi* kích thước cũ.

+ *shrink*: trong tác vụ pop(), khi kiểm tra thấy mảng đầy 25%, ta thu nhỏ kích thước mảng bằng cách cấp phát mảng mới có kích thước phân nửa kích thước cũ. Nguyên tắc này không cài đặt trong ví dụ dưới.

```
public class ArrayStack<E> {
    private E[] storage;
    private int size = 0;
    private static final int INITIAL_CAPACITY = 100;

    public ArrayStack(int capacity) {
        storage = (E[]) new Object[capacity];
    }

    public ArrayStack() {
        this(INITIAL_CAPACITY);
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public void push(E element) {
        if (size == storage.length) {
            resize();
        }
        storage[size++] = element;
    }

    public E topEl() {
        if (size == 0) throw new java.util.EmptyStackException();
        return storage[size - 1]; // đỉnh stack (top) có chỉ số size - 1
    }

    public E pop() {
        if (isEmpty()) throw new java.util.EmptyStackException();
        E element = storage[--size];
        storage[size] = null;
        return element;
    }

    @Override public String toString() {
        return String.format("Top: %d%n%s%n", size - 1, java.util.Arrays.toString(storage));
    }

    private void resize() {
        Object[] tmp = new Object[2 * storage.length]; // repeated doubling
        System.arraycopy(storage, 0, tmp, 0, size);
        storage = (E[]) tmp;
    }
}
```

2. Ứng dụng

Stack là cấu trúc dữ liệu được dùng phổ biến, hỗ trợ cho nhiều giải thuật: duyệt theo chiều sâu (Depth-First Traversal), backtracking, xử lý biểu thức biểu thức postfix RPN (Polish Notation Reverse), nhiều vấn đề trong hệ điều hành, trình biên dịch, khử đệ quy, ... Một số vấn đề được giải quyết bằng cách sử dụng stack, được trình bày sau đây:

a) So khớp dấu ngoặc

Các dấu ngoặc phải so khớp về loại, tránh trường hợp các cặp ngoặc thay vì lồng nhau lại chéo nhau, như ví dụ: `({ })`. Các dấu ngoặc cũng phải so khớp về số lượng, tránh trường hợp số ngoặc mở nhiều hơn hoặc ít hơn số ngoặc đóng.

Giải thuật

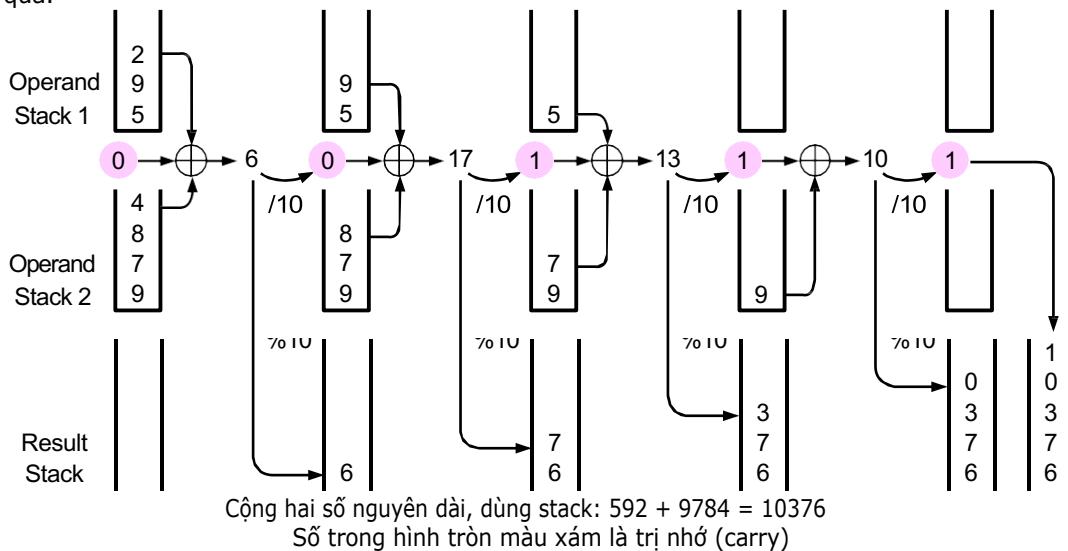
- Duyệt chuỗi nhập. Nếu gặp các dấu mở ngoặc thì đưa (push) chúng vào stack.
- Nếu gặp dấu đóng ngoặc thì lấy (pop) dấu mở ngoặc tại đỉnh stack ra so sánh. Nếu chúng khớp nhau (cùng cặp) thì tiếp tục duyệt; nếu không thông báo lỗi và kết thúc.
- Kết thúc duyệt khi đọc hết chuỗi nhập. Nếu stack rỗng, so khớp thành công; nếu không, thông báo lỗi.

b) Phép cộng số nguyên dài

Số nguyên dài không được lưu với các kiểu dữ liệu số, chúng được lưu dưới dạng chuỗi các ký tự số.

Giải thuật

- Dùng 3 stack: Operand Stack 1 dùng chứa chuỗi số nguyên dài thứ nhất, Operand Stack 2 dùng chứa chuỗi số nguyên dài thứ hai, Result Stack chứa số nguyên dài là kết quả của phép cộng. Trị nhớ (carry) khởi tạo bằng 0.
- Đưa số nguyên dài thứ nhất vào Operand Stack 1, đưa số nguyên dài thứ hai vào Operand Stack 2.
 - Lấy các phần tử trên đỉnh Operand Stack 1 và Operand Stack 2 cộng nhau và cộng với carry được kết quả n. Đưa ($n \% 10$) vào Result Stack, carry mới bằng ($n / 10$).
 - Nếu Operand Stack 1 rỗng trước, lấy phần tử trên đỉnh Operand Stack 2 cộng với carry được kết quả n. Đưa ($n \% 10$) vào Result Stack, carry mới bằng ($n / 10$). Thực hiện tương tự nếu Operand Stack 2 rỗng trước.
 - Nếu Operand Stack 1 và Operand Stack 2 đều rỗng, đưa carry cuối cùng vào Result Stack, lấy toàn bộ Result Stack đưa vào chuỗi kết quả.



```
import java.util.Stack;
```

```
public class BigInteger {
    protected String number;

    public BigInteger(String number) {
        this.number = number;
    }

    private String stackToString(Stack<Integer> stack) {
        String s = "";
        while (!stack.isEmpty()) s += stack.pop();
        return s;
    }

    private Stack<Integer> stringToStack(String s) {
        Stack<Integer> stack = new Stack<>();
        for (int i = 0; i < s.length(); ++i)
            stack.push((int) (s.charAt(i) - '0'));
        return stack;
    }

    public BigInteger add(BigInteger other) {
        int num1, num2;
        int carry = 0;
```

```

Stack<Integer> opStack1 = stringToStack(this.number);
Stack<Integer> opStack2 = stringToStack(other.number);
Stack<Integer> resultStack = new Stack<>();
while (!opStack1.isEmpty() || !opStack2.isEmpty()) {
    num1 = (!opStack1.isEmpty() ? opStack1.pop() : 0) + carry;
    num2 = !opStack2.isEmpty() ? opStack2.pop() : 0;
    resultStack.push((num1 + num2) % 10);
    carry = (num1 + num2) / 10;
}
if (carry > 0) // trị carry cuối
    resultStack.push(carry);
return new BigInteger(stackToString(resultStack));
}

@Override public String toString() {
    return number;
}
}
}

```

c) Từ biểu thức infix chuyển thành postfix

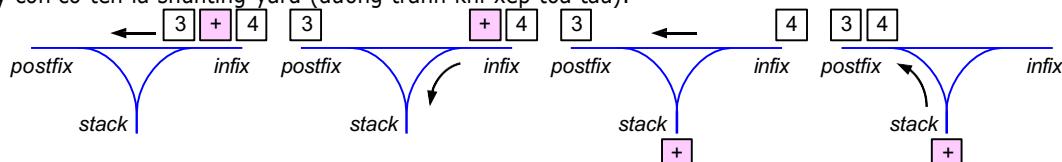
Giải thuật (Dijkstra, 1960)

- Duyệt biểu thức infix từ trái sang phải.
- Nếu gặp toán hạng, ghi vào chuỗi kết quả.
- Nếu gặp dấu mở ngoặc (, đưa vào stack.
- Nếu gặp toán tử op1:
 - + Chừng nào độ ưu tiên của toán tử op1 còn *nhỏ hơn hoặc bằng* độ ưu tiên của toán tử op2 trên đỉnh stack. Lấy op2 ra khỏi stack và ghi vào chuỗi kết quả.
 - + Đưa op1 vào stack.
- Nếu gặp dấu đóng ngoặc), lấy tất cả các toán tử trong stack ra và ghi vào chuỗi kết quả cho đến khi lấy được dấu mở ngoặc (ra khỏi stack.
- Khi duyệt hết biểu thức infix, lấy tất cả các toán hạng nếu còn từ stack và ghi vào chuỗi kết quả.

Ví dụ: infix (80 - 30) * (40 + 50 / 10), postfix 80 30 - 40 50 10 / + *

Đọc infix	Xử lý	Stack	Biểu thức postfix kết quả
(push "("	(
80	xuất "80"	(80
-	push "-"	(-	80
30	xuất "30"	(-	80 30
)	pop và xuất "-", pop "("		80 30 -
*	push "*"	*	80 30 -
(push "("	* (80 30 -
40	xuất "40"	* (80 30 - 40
+	push "+"	* (+	80 30 - 40
50	xuất "50"	* (+	80 30 - 40 50
/	push "/" ưu tiên hơn "+" tại đỉnh stack	* (+ /	80 30 - 40 50
10	xuất "10"	* (+ /	80 30 - 40 50 10
)	pop và xuất "/" rồi "+", pop "("	*	80 30 - 40 50 10 / +
Kết thúc	pop "*"	rỗng	80 30 - 40 50 10 / + *

Giải thuật này còn có tên là shunting-yard (đường tránh khi xếp toa tàu):



Trong hình trên, mỗi toa tàu tương ứng với một phần tử trong biểu thức infix. Có ba cách chuyển toa tàu tương ứng với các luật mô tả trong giải thuật: từ infix sang postfix, từ infix vào "đường tránh" stack và từ stack ra postfix.

```

private int priority(String s) {
    if ("+-".contains(s)) return 0;
    if ("*/".contains(s)) return 1;
    return -1;
}

public void infix2postfix(String infix) {
    Stack<String> stack = new Stack<>();
    Scanner scanner = new Scanner(infix);
    while (scanner.hasNext()) {
        if (scanner.hasNextInt()) {
            System.out.print(scanner.nextInt() + " ");
        } else {
    }
}

```

```

String str = scanner.next();
if ("+-*/".contains(str)) {
    while (!stack.empty() && priority(str) <= priority(stack.peek()))
        System.out.print(stack.pop() + " ");
    stack.push(str);
} else if (str.equals("(")) {
    stack.push(str);
} else if (str.equals(")")) {
    String s;
    do {
        s = stack.pop();
        if (!s.equals("(")) System.out.print(s + " ");
    } while (!s.equals("("));
}
}
while (!stack.isEmpty()) System.out.print(stack.pop() + " ");
System.out.println();
}

```

d) Định trị biểu thức postfix

Giải thuật

- Duyệt biểu thức postfix từ trái sang phải.
- + Nếu gặp toán hạng, đưa toán hạng đó vào stack.
- + Nếu gặp toán tử op, lấy hai toán hạng y và x lần lượt ra khỏi stack, tính ($x \text{ op } y$) và đưa kết quả vào lại stack.
- Khi duyệt hết biểu thức postfix, trị cuối cùng trong stack là giá trị của biểu thức postfix.

```

public double postfixEval(String postfix) {
    Stack<Double> stack = new Stack<>();
    Scanner scanner = new Scanner(postfix);
    while (scanner.hasNext()) {
        if (scanner.hasNextDouble()) {
            stack.push(scanner.nextDouble());
        } else {
            String input = scanner.next();
            char op = input.charAt(0);
            double y = stack.pop();
            double x = stack.pop();
            double z = 0;
            switch (op) {
                case '+': z = x + y; break;
                case '-': z = x - y; break;
                case '*': z = x * y; break;
                case '/': z = x / y;
            }
            stack.push(z);
        }
    }
    return stack.pop();
}

```

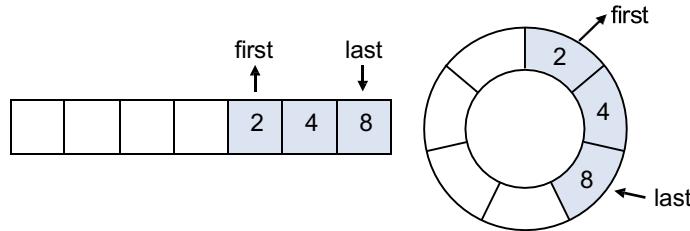
II. Queue

Queue (hàng đợi) là một cấu trúc dữ liệu có *hai đầu*, các phần tử được đưa vào queue từ một đầu (last) và lấy ra từ đầu còn lại (first). Phần tử đưa vào đầu tiên sẽ ra khỏi queue trước hết, đó là nguyên tắc hoạt động FIFO (First In First Out) của queue. Các phần tử trong queue sẽ được lấy ra theo đúng thứ tự khi đưa chúng vào queue. Queue được cài đặt bằng mảng hoặc bằng danh sách liên kết.

1. Các thao tác với queue

- **isEmpty()**: tương đương với việc kiểm tra danh sách liên kết có rỗng hay không.
- **enqueue()**: (entering the queue) độ phức tạp O(1). Đưa thêm phần tử vào queue tương đương với *chèn cuối* vào danh sách liên kết. Nếu cài đặt bằng mảng, chú ý ném exception khi đưa một phần tử vào queue đã đầy, gây lỗi *queue overflow*.
- **dequeue()**: (deleting from the queue) độ phức tạp O(1). Lấy phần tử ra khỏi queue tương đương với *xóa đầu* danh sách liên kết. Chú ý ném exception khi lấy phần tử ra khỏi một queue đang rỗng, gây lỗi *queue underflow*.
- **firstEl()**: trả về phần tử tại ngõ ra (first) của queue mà không cần lấy nó ra khỏi queue.
- **clear()**: xóa toàn bộ queue, làm rỗng queue.

Nếu cài đặt bằng mảng, các phần tử được lấy ra từ đầu mảng (first) và thêm vào từ cuối mảng (last). Vì last luôn tăng nên sẽ dẫn đến hết ô trống. Để khắc phục điều này, ta thường ghép cuối mảng vào đầu mảng để tạo mảng vòng (circular array), khi đó các ô trống đầu mảng sẽ được sử dụng để nhận phần tử vào.



Trái: cài đặt queue bằng mảng, không còn ô trống tại last để enqueue dù có nhiều ô trống tại first.

Phải: cài đặt queue bằng mảng vòng, các ô trống được sử dụng linh hoạt.

Cài đặt queue bằng mảng:

```
public class ArrayQueue<E> {
    private E[] storage;
    public int first = -1;
    public int last = -1;
    private int size = 0;
    private static final int INITIAL_CAPACITY = 7;

    public ArrayQueue(int capacity) {
        storage = (E[]) new Object[capacity];
    }

    public ArrayQueue() {
        this(INITIAL_CAPACITY);
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public void enqueue(E element) {
        if (size == storage.length) {
            resize();
        }
        if (size == 0) {
            first = 0;
        }
        last = (last + 1) % storage.length;
        storage[last] = element;
        size++;
    }

    public E firstEl() {
        if (isEmpty()) {
            throw new java.util.NoSuchElementException();
        }
        return storage[first];
    }

    public E dequeue() {
        if (isEmpty()) {
            throw new java.util.NoSuchElementException();
        }
        E element = storage[first];
        storage[first] = null;
        size--;
        if (isEmpty()) {
            first = last = -1;
        } else {
            first = (first + 1) % storage.length;
        }
        return element;
    }

    @Override
    public String toString() {
        return String.format("First: %d\nLast : %d\n%s\n", first, last, java.util.Arrays.toString(storage));
    }

    private void resize() {
```

```

Object[] tmp = new Object[2 * storage.length];
System.arraycopy(storage, 0, tmp, 0, size);
storage = (E[]) tmp;
}
}

```

2. Ứng dụng

Queue rất hữu dụng trong những vấn đề có duyệt theo chiều rộng (Breadth-First Traversal), xử lý dữ liệu bằng bộ đệm, lập lịch trong hệ điều hành, ...

3. Priority Queue (hàng đợi ưu tiên)

Thông thường, ta lấy phần tử khỏi queue theo nguyên tắc FIFO. Với priority queue, phần tử được lấy ra phụ thuộc vào *độ ưu tiên* (theo một tiêu chí nào đó) và *vị trí hiện tại* của nó, không theo nguyên tắc FIFO.

Ví dụ, phần tử được lấy ra là phần tử nhỏ nhất trong các phần tử hiện có trong queue; nếu có hai phần tử nhỏ nhất (bằng nhau), lấy phần tử gần đầu first nhất.

Có nhiều giải pháp cài đặt cho hàng đợi ưu tiên:

- Danh sách liên kết được sắp xếp theo độ ưu tiên

Khi enqueue, cần duy trì thứ tự bằng cách đặt phần tử mới vào vị trí được xác định theo độ ưu tiên của nó. Tương đương với thao tác tìm kiếm và chèn tại vị trí chỉ định vào danh sách liên kết, độ phức tạp O(n).

Khi dequeue, vì danh sách đã có thứ tự nên thao tác lấy ra tương đương xóa phần tử của danh sách liên kết, độ phức tạp O(1).

- Danh sách liên kết không sắp xếp thứ tự theo độ ưu tiên

Khi enqueue ta đơn giản chèn phần tử mới vào, tương đương chèn phần tử vào danh sách liên kết, độ phức tạp O(1).

Khi dequeue, ta tìm phần tử theo độ ưu tiên và lấy nó ra khỏi danh sách. Tương đương với thao tác tìm kiếm và xóa phần tử tìm được ra khỏi danh sách liên kết, độ phức tạp O(n).

Có thể duy trì một vùng đệm (cache) chứa một số phần tử ưu tiên nhất, cache được hiệu chỉnh khi enqueue/dequeue.

- Hàng đợi đa cấp

Nếu độ ưu tiên có các cấp xác định, dùng mảng các queue thông thường, mỗi queue chứa các phần tử có cùng cấp ưu tiên.

- Heap

Heap là một cấu trúc dữ liệu sẽ được thảo luận trong phần cây nhị phân, bản chất là một priority queue cài đặt bằng mảng. Khi có phần tử mới đưa vào, heap sẽ cấu trúc lại sao cho phần tử trên đỉnh heap (đầu mảng) luôn lớn nhất hoặc nhỏ nhất. Nếu phần tử trên đỉnh heap lớn nhất, ta có maxheap; nếu phần tử trên đỉnh heap nhỏ nhất, ta có minheap.

Ngoài priority queue, một dạng queue đặc biệt thường được dùng là deque (double-ended queue). Deque là một dạng mở rộng của queue, cho phép lấy ra và đưa vào phần tử từ cả hai đầu queue. Như vậy các tác vụ của deque gồm:

- isEmpty (kiểm tra deque rỗng) và clear (xóa deque).
- Tại đầu first: enqueueFirst, dequeueFirst, firstEl.
- Tại đầu last: enqueueLast, dequeueLast, lastEl.

Đệ quy

Đệ quy là một phương pháp lập trình được sử dụng phổ biến để giải các bài toán được định nghĩa đệ quy, các bài toán sử dụng chiến lược "chia để trị", các bài toán dạng tìm kiếm quay lui.

I. Khái niệm

1. Định nghĩa đệ quy

Cách định nghĩa một bài toán dựa trên chính bài toán đó với *quy mô thu nhỏ* gọi là định nghĩa đệ quy. Định nghĩa như vậy không dẫn đến lặp vô hạn do bài toán với quy mô thu nhỏ tại một mức nào đó có *lời giải xác định*.

Như vậy, định nghĩa đệ quy bao gồm hai phần:

- Trường hợp cơ sở (anchor case, trường hợp gốc): là trường hợp định nghĩa đệ quy trả về một kết quả xác định.
- Trường hợp đệ quy (recursive case): là biểu thức truy hồi mô tả luật đệ quy.

Chúng ta có thể dùng cách tiếp cận như sau để viết một hàm đệ quy:

1. Định nghĩa chính xác vấn đề cần giải quyết. Tốt nhất là trình bày nó dưới dạng một biểu thức truy hồi.
2. Xác định quy mô (kích thước) của vấn đề. Kích thước này sẽ được truyền như đối số đến hàm gọi và sẽ thay đổi (thường là thu nhỏ) sau mỗi lần gọi hàm.
3. Xác định và giải quyết trường hợp cơ sở, nghĩa là trường hợp mà vấn đề có thể được giải quyết không đệ quy, thường là điều kiện đầu (initial condition) của biểu thức truy hồi. Đây chính là điểm dừng đệ quy, nếu không định nghĩa, hàm đệ quy sẽ thực hiện đến khi tràn stack.
4. Xác định và giải quyết trường hợp tổng quát, thường dựa vào biểu thức truy hồi hoặc tính chất đệ quy của vấn đề. Giải quyết trường hợp tổng quát theo cách đệ quy thường là thu nhỏ quy mô vấn đề để chuyên dần nó về trường hợp cơ sở.

Ví dụ: Giả sử dân số thế giới năm 2000 là 8 tỷ người với mức tăng trưởng hàng năm không đổi là 1.8%. Viết hàm đệ quy tính dân số thế giới năm bất kỳ sau năm 2000.

Lời giải:

5. Biểu thức truy hồi:

$$P_y = \begin{cases} 8 \cdot 10^9 & y = 2000 \\ P_{y-1} + \frac{1.8}{100} P_{y-1} & \text{otherwise} \end{cases}$$

2. Kích thước đệ quy: năm (year), sẽ giảm cho mỗi lần gọi đệ quy.

3. Trường hợp cơ sở: giải quyết dựa vào điều kiện đầu: $P_{2000} = 8 \cdot 10^9$

4. Trường hợp tổng quát: giải quyết dựa vào biểu thức truy hồi: $P_y = P_{y-1} + P_{y-1} * 1.8\% = P_{y-1} * (1 + 0.018)$

Từ phân tích trên ta dễ dàng viết được hàm đệ quy giải quyết vấn đề:

```
double population(int year) {
    if (year == 2000) return 8E9;
    return population(year - 1) * (1 + 0.018);
}
```

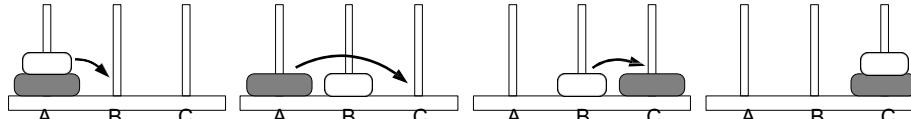
Quy nạp (induction) là một khái niệm dễ nhầm lẫn với đệ quy (recursion). Với đệ quy, bắt đầu từ trường hợp tổng quát, ta quy về trường hợp cơ sở. Với quy nạp, bắt đầu từ trường hợp cơ sở, ta phát triển thành trường hợp tổng quát.

Đệ quy thường được sử dụng do định nghĩa đệ quy trực quan, dễ đọc, gần gũi với cách tư duy của con người. Trong đa số trường hợp, mã cài đặt đệ quy ngắn hơn so với cài đặt không đệ quy.

Ví dụ: Bài toán tháp Hanoi (Towers of Hanoi) rất khó giải nếu không định nghĩa đệ quy.

Có 3 cọc A, B, C; khởi đầu cọc A có n đĩa, sắp xếp sao cho đĩa lớn hơn luôn nằm bên dưới, 2 cọc kia trống. Hãy chuyển tất cả đĩa từ cọc A sang cọc C, được dùng cọc phụ B. Trong quá trình chuyển phải đảm bảo đĩa lớn hơn luôn nằm bên dưới.

Minh họa với 2 đĩa:



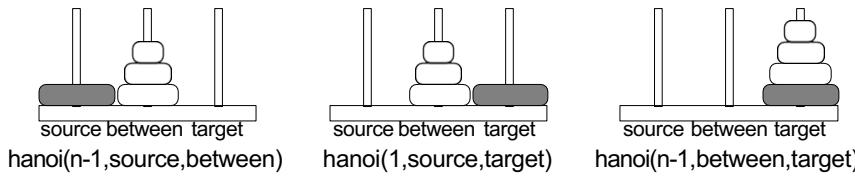
Gọi `hanoi(n, source, target)` là phép chuyển n đĩa từ cọc `source` sang cọc `target`. Đặt trị cho từng cọc: `source(1)`, `between(2)`, `target(3)`. Ta có: `source + between + target = 6`. Từ đó dễ dàng xác định được cọc `between` trong các lời gọi đệ quy khác (với `source` và `target` khác): `between = 6 - (source + target)`.

Trường hợp tổng quát: giả sử ta đã chuyển được $n - 1$ đĩa (trừ đĩa lớn nhất) từ cọc `source` sang cọc trung gian `between` bằng phép chuyển `hanoi(n - 1, source, between)`, thì ta có thể dễ dàng thực hiện được `hanoi(n, source, target)` bằng các thao tác:

- `hanoi(n - 1, source, between)`: chuyển $n - 1$ đĩa (trừ đĩa lớn nhất) từ cọc `source` sang cọc trung gian `between`.

- `hanoi(1, source, between)`: chuyển (một) đĩa thứ n từ cọc `source` sang cọc `target`.

- `hanoi(n - 1, between, target)`: chuyển $n - 1$ đĩa từ cọc trung gian `between` sang cọc `target`, chồng lên đĩa lớn nhất vừa chuyển sang.



Định nghĩa đệ quy cho `hanoi(n, source, target)`

Trường hợp cơ sở: là điều kiện đầu để dừng đệ quy. Nếu chỉ có một đĩa, ta đơn giản chuyển đĩa này trực tiếp từ cọc `source` sang cọc `target`.

```
void hanoi(int n, int source, int target) {
```

```

int between = 6 - (source + target);
if (n == 1) System.out.printf("Disk %d: [%d] -> [%d]%n", n, source, target);
else {
    hanoi(n - 1, source, between);
    System.out.printf("Disk %d: [%d] -> [%d]%n", n, source, target);
    hanoi(n - 1, between, target);
}
}

```

2. Phân tích lời gọi đệ quy

Hệ thống sẽ lưu trạng thái của lời gọi hàm vào một bản ghi hoạt động (Activation Record) rồi đặt vào run-time stack. Mỗi Activation Record chứa tri trả về, địa chỉ trả về, các tham số truyền cho hàm, các biến cục bộ của mỗi lời gọi hàm. Activation Record được lấy ra khỏi run-time stack khi hàm được gọi kết thúc các lệnh của nó.

Khi gọi đệ quy, hàm được gọi lại gọi tiếp hàm khác, các Activation Record của các hàm được gọi lần lượt chồng lên nhau trong run-time stack theo thứ tự gọi. Nếu có sai sót trong thiết kế chương trình đệ quy, chuỗi lời gọi hàm không có điểm dừng, các Activation Record chồng lên nhau liên tục làm run-time stack tràn, gây lỗi stack overflow.

Với bài toán có quy mô lớn, đệ quy nhiều lần, phải tốn vùng nhớ để lưu trữ tạm các Activation Record. Ngoài ra, việc tạo Activation Record, đặt vào và đẩy ra khỏi run-time stack sinh thêm thời gian stack overhead, làm cho đệ quy hoạt động chậm. Đây là các hạn chế của đệ quy.

Bản chất của đệ quy là dùng run-time stack, nên ta có thể khử đệ quy bằng cách dùng một stack tường minh, thay thế cho run-time stack.

II. Phân loại đệ quy

1. Tail Recursive

Trong đệ quy đuôi, hay đệ quy tầm thường, phát biểu đệ quy được thực hiện cuối cùng. Không có tác vụ nào được thực hiện sau lời gọi đệ quy.

```

long factorial(int n) {
    if (n == 0) return 1;
    return (n * factorial(n - 1));
}

```

Có thể khử đệ quy đuôi bằng cách chuyển thành vòng lặp.

```

long factorial(int n) {
    long f = 1;
    for (int i = 1; i <= n; ++i)
        f *= i;
    return f;
}

```

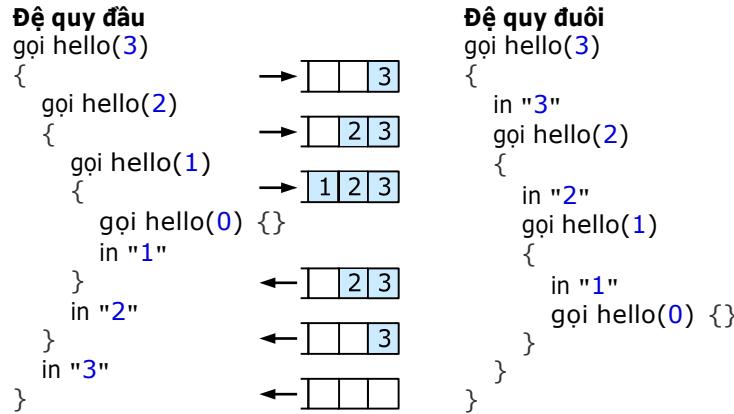
2. Nontail Recursive

Lời gọi đệ quy được gọi trước các tác vụ trong một lần đệ quy, còn gọi là đệ quy đầu. Khi kết thúc đệ quy, biến cục bộ được đẩy ra khỏi stack theo nguyên tắc LIFO (Last In First Out - vào sau ra trước) và được sử dụng, nên kết quả xuất ngược với thứ tự dùng các biến cục bộ. Đặc tính này được sử dụng trong backtracking.

```

void hello(int n) {
    if (n == 0) return;
    hello(n - 1);           // gọi đệ quy
    System.out.printf("%d", n); // tác vụ trong một lần đệ quy
}

```



Trái: đệ quy đầu, các Activation Record lưu trữ của lời gọi đệ quy để in sau. Phải: đệ quy đuôi.

3. Indirect Recursion

Đệ quy gián tiếp (indirect recursion): chuỗi gọi hàm dẫn đến việc gọi lại một hàm trong chuỗi gọi, ví dụ: f1 gọi f2, f2 gọi f3, f3 gọi f1. Ta còn gọi là "vào lai" (re-entrant).

Đệ quy tương hỗ (mutual recursion): hai hàm gọi nhau một cách đệ quy, f1 gọi f2 và f2 gọi lại f1 với quy mô nhỏ hơn. Cũng giống như đệ quy thông thường, cách đệ quy này yêu cầu mỗi hàm gọi có trường hợp cơ sở (có điểm dừng) và kích thước bài toán thay đổi (thường là giảm) sau mỗi lời gọi đệ quy.

Ví dụ:

Viết các hàm đệ quy tính $\sin(x)$ và $\cos(x)$ theo cặp đồng nhất thức:

$$\begin{cases} \sin 3x = (4\cos^2 x - 1)\sin x \\ \cos 3x = (1 - 4\sin^2 x)\cos x \end{cases}$$

và cặp đồng nhất thức xấp xỉ (khai triển Taylor) cho các giá trị rất nhỏ của x ($x < 5.10^{-4}$):

$$\begin{cases} \sin x \approx x - x^3/6 \\ \cos x \approx 1 - x^2/2 \end{cases}$$

Với độ chính xác 5.10^{-4} .

Phân tích:

1. Tính $\sin(x)$ và $\cos(x)$ dựa vào đệ quy tương hỗ.
2. Kích thước đệ quy: trị x , sẽ giảm $1/3$ sau mỗi lần gọi đệ quy.
3. Trường hợp cơ sở: với x rất nhỏ ($x < 5.10^{-4}$) ta không dùng cặp đồng nhất thức đệ quy tương hỗ để tính mà dùng cặp đồng nhất thức xấp xỉ không đệ quy để tính tường minh trị của $\sin x$ và $\cos x$:

$$\begin{cases} \sin x \approx x - x^3/6 \\ \cos x \approx 1 - x^2/2 \end{cases}$$

và như vậy ta dừng luôn đệ quy tại đây.

4. Trường hợp tổng quát: với x lớn ta dùng cặp đồng nhất thức đệ quy tương hỗ để giảm dần kích thước bài toán:

$$\begin{cases} \sin 3x = (4\cos^2 x - 1)\sin x \\ \cos 3x = (1 - 4\sin^2 x)\cos x \end{cases}$$

Chú ý kích thước đệ quy giảm cho mỗi lần gọi đệ quy, đây là điều kiện để chuyển về trường hợp cơ sở.

```
public double SIN(double x) {
    if (Math.abs(x) < eps) return x * (1 - x * x / 6);
    return (4 * COS(x / 3) * COS(x / 3) - 1) * SIN(x / 3);
}

public double COS(double x) {
    if (Math.abs(x) < eps) return (1 - x * x / 2);
    return (1 - 4 * SIN(x / 3) * SIN(x / 3)) * COS(x / 3);
}
// kiểm thử với lớp Math của Java
System.out.println(SIN(Math.PI/180));
System.out.println(Math.sin(Math.PI/180));
```

4. Nested Recursion

Đệ quy lồng, trong đó đối số của hàm đệ quy lại là hàm đệ quy, điều này dẫn đến số lời gọi đệ quy tăng lên rất nhanh.

Hàm Ackermann là ví dụ điển hình về hàm đệ quy không cơ bản, có trị tăng cực nhanh. Ví dụ dạng thập phân của $A(4, 3)$ không trình bày được vì có số chữ số lớn hơn số nguyên tử ước lượng của vũ trụ (10^{80}).

Hàm Ackermann được hiện thực bằng cách dùng biểu thức truy hồi sau:

$$A(n,m) = \begin{cases} m+1 & n=0 \\ A(n-1,1) & m=0 \\ A(n-1, A(n,m-1)) & n,m > 0 \end{cases}$$

Điều kiện đầu: $A(0, m) = m + 1$

$A(3, 6)$ gọi đệ quy 172233, lồng sâu 511 cấp, thường dùng đo tốc độ hoạt động (benchmark).

```
int A(int n, int m) {
    if (n == 0) return m + 1;
    if (m == 0) return A(n - 1, 1);
    return A(n - 1, A(n, m - 1));
}
```

5. Excessive Recursion

Đệ quy quá sâu hay rẽ nhánh quá nhiều làm tăng nhanh số lời gọi đệ quy, có thể dẫn đến tràn run-time stack và phá hủy chương trình. Rẽ nhánh ở đây được hiểu theo nghĩa: một lời gọi đệ quy gọi *nhiều lời gọi đệ quy khác*.

Điển hình là hàm Fibonacci đệ quy:

$$F_n = \begin{cases} 1 & n=1,2 \\ F_{n-1} + F_{n-2} & n > 2 \end{cases}$$

```
int Fi(int n) {
    return (n < 3) ? 1 : Fi(n - 1) + Fi(n - 2);
}
```

Vì vậy, khi số lời gọi và thời gian thực hiện tăng theo hàm mũ thì ta không nên dùng giải thuật đệ quy. Ví dụ trong trường hợp tính Fibonacci, ta thường thay thế bằng giải thuật quy hoạch động, bằng vòng lặp hoặc dùng công thức dạng đóng (của A. de Moivre) dùng tính số Fibonacci thứ n như sau:

$$F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}} \quad \varphi = \frac{1+\sqrt{5}}{2}, \psi = 1-\varphi$$

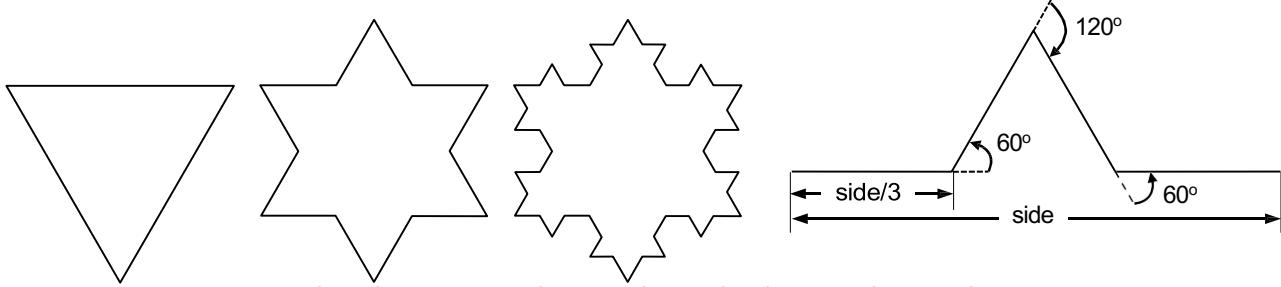
φ gọi là hằng số vàng (golden mean).

Giải pháp dùng vòng lặp:

```
int Fi(int n) {
    int fi, fi1, fi2;
    fi = fi1 = fi2 = 1;
    for (int i = 3; i <= n; ++i) {
        fi = fi1 + fi2;
        fi1 = fi2;
        fi2 = fi;
    }
    return fi;
}
```

III. Đệ quy và hình học phân hình

Những mẫu hình học mang tính đệ quy thường được thấy trong tự nhiên. Ví dụ hình dạng bông tuyết, được nhà toán học Helge von Koch mô tả (1904). Các mẫu này gọi là mẫu hình học phân hình (fractal geometry), chúng được định nghĩa bởi một quá trình đệ quy.



Trái: đường von Koch, áp dụng liên tục lên các cạnh một tam giác.
Phải: mô tả thủ tục vẽ 4 đường (draw4Lines).

Đường von Koch K_0 là một đoạn thẳng nằm ngang có chiều dài $side$. Ta tạo K_1 bằng thao tác vẽ 4 đường:

- chia cạnh $side$ thành ba phần bằng nhau, cạnh này có góc φ . Chú ý góc φ ngược chiều kim đồng hồ là góc dương (+).
- vẽ cạnh kích thước $side/3$ theo hướng chỉ định bởi φ , trong chương trình φ khởi tạo bằng 0.
- quay trái 60° (thay đổi $\varphi += 60^\circ$) và vẽ cạnh kích thước $side/3$ theo hướng chỉ định bởi φ .
- quay phải 120° (thay đổi $\varphi -= 120^\circ$) và vẽ cạnh kích thước $side/3$ theo hướng chỉ định bởi φ .
- quay trái 60° (thay đổi $\varphi += 60^\circ$) và vẽ cạnh kích thước $side/3$ theo hướng chỉ định bởi φ .

K_2 được tạo bằng cách thực hiện thao tác vẽ 4 đường trên 4 đoạn của K_1 .

Định nghĩa đệ quy: đường von Koch bậc level được tạo ra bởi 4 đường von Koch bậc level - 1.

Draw4Lines(side, level)

```
    nếu (level == 0)
        vẽ một đoạn thẳng;
    ngược lại
        draw4Lines(side/3, level - 1)
        quay trái 60°
        draw4Lines(side/3, level - 1)
        quay phải 120°
        draw4Lines(side/3, level - 1)
        quay trái 60°
        draw4Lines(side/3, level - 1)
```

Khi vẽ một đoạn thẳng, nếu gốc của hệ tọa độ cực là đỉnh đi, tọa độ đỉnh tới của đoạn thẳng là $P(R, \varphi)$. Trong đó, φ là góc cực, R là bán kính cực. Tọa độ cực $P(R, \varphi)$ được chuyển thành tọa độ Descartes $P(x, y)$ khi vẽ ra màn hình:

$$P \begin{cases} x = x_0 + R \cos \varphi \\ y = y_0 + R \sin \varphi \end{cases}$$

(x_0, y_0) là gốc hệ tọa độ cực, cũng là đỉnh đi của đoạn thẳng đó.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class Koch extends JFrame {
    public Koch() {
        setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        setSize(600, 500);
        add(new CurveKoch());
    }

    public static void main(String[] args) {
```

```

        new Koch().setVisible(true);
    }

}

class CurveKoch extends Canvas {
    public float x, y;
    float phi;
    int midX, midY, level = 1;

    int iX(float x) { return Math.round(midX + x); }
    int iY(float y) { return Math.round(midY - y); }

    CurveKoch() {
        addMouseListener(new MouseAdapter() {
            @Override public void mousePressed(MouseEvent evt) {
                level++; // click mouse để tăng level
                repaint();
            }
        });
    }

    @Override public void paint(Graphics g) {
        Dimension d = getSize();
        int maxX = d.width - 1, maxY = d.height - 1, length = 3 * maxX / 4;
        midX = maxX / 2;
        midY = maxY / 2;
        x = (float) (-length / 2); // điểm khởi đầu
        y = 0;
        phi = 0;
        drawKoch(g, length, level);
    }

    public void drawKoch(Graphics g, float side, int level) {
        if (level == 0) {
            float x1 = x + side * (float) Math.cos(phi * Math.PI / 180);
            float y1 = y + side * (float) Math.sin(phi * Math.PI / 180);
            g.drawLine(iX(x), iY(y), iX(x1), iY(y1));
            x = x1;
            y = y1;
        } else {
            drawKoch(g, side/3, level - 1);
            phi += 60;
            drawKoch(g, side/3, level - 1);
            phi -= 120;
            drawKoch(g, side/3, level - 1);
            phi += 60;
            drawKoch(g, side/3, level - 1);
        }
    }
}

```

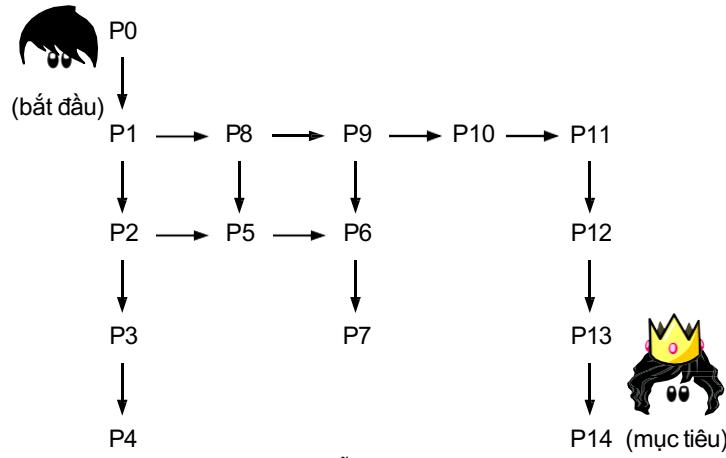
IV. Backtracking

Trong nhiều vấn đề thực tế, để đạt được giải pháp cần tìm kiếm vét cạn một tập lớn các khả năng có thể. Để thu giảm không gian tìm kiếm, cần thiết phải phát triển các kỹ thuật tìm kiếm có hệ thống. Phần này giới thiệu kỹ thuật backtracking (quay lui), thường được mô tả như một kỹ thuật *tìm kiếm vét cạn có hệ thống* nhằm tránh việc duyệt tất cả các khả năng.

Ý tưởng của backtracking: Ta khám phá từng con đường có thể dẫn đến giải pháp, chọn một quyết định tại một thời điểm và ngay khi ta thấy con đường đang chọn không dẫn đến giải pháp, quay lui trở lại bước quyết định trước gần nhất. Tại bước quay lui đó, ta sẽ khám phá những con đường khác, nếu có. Nếu không có con đường nào, ta lại quay lui bước nữa. Cứ như thế cho đến khi đạt được giải pháp. Nếu ta quay lui trở về bước đầu mà không tìm thấy bất kỳ con đường nào dẫn đến giải pháp, bài toán không có giải pháp.

Khác với tìm kiếm vét cạn (brute force), kỹ thuật quay lui sớm loại bỏ phần đường đi không dẫn đến giải pháp, nó không mù quáng kiểm tra chúng đến cùng như tìm kiếm vét cạn.

Xem ví dụ dưới, ta bắt đầu từ vị trí P0 và muốn tìm đường đến vị trí mục tiêu P14 (giải pháp). Tại một vị trí, chiến lược thử là: từ trên xuống dưới, từ trái sang phải. Chuỗi thử P0-P1-P2-P3-P4 không còn vị trí thử mà vẫn chưa thấy mục tiêu; ta quay lại vị trí P3, kết quả vẫn thất bại. Ta quay lại vị trí P2, chuyển sang chuỗi thử P2-P5-P6-P7 vẫn thất bại. Ta quay lại P1 và tìm được chuỗi thử đi đến mục tiêu P1-P8-P9-P10-P11-P12-P13-P14. Chú ý, không thử lại P5 từ P8, P6 từ P9 vì P5 và P6 đã được thăm trước đó.

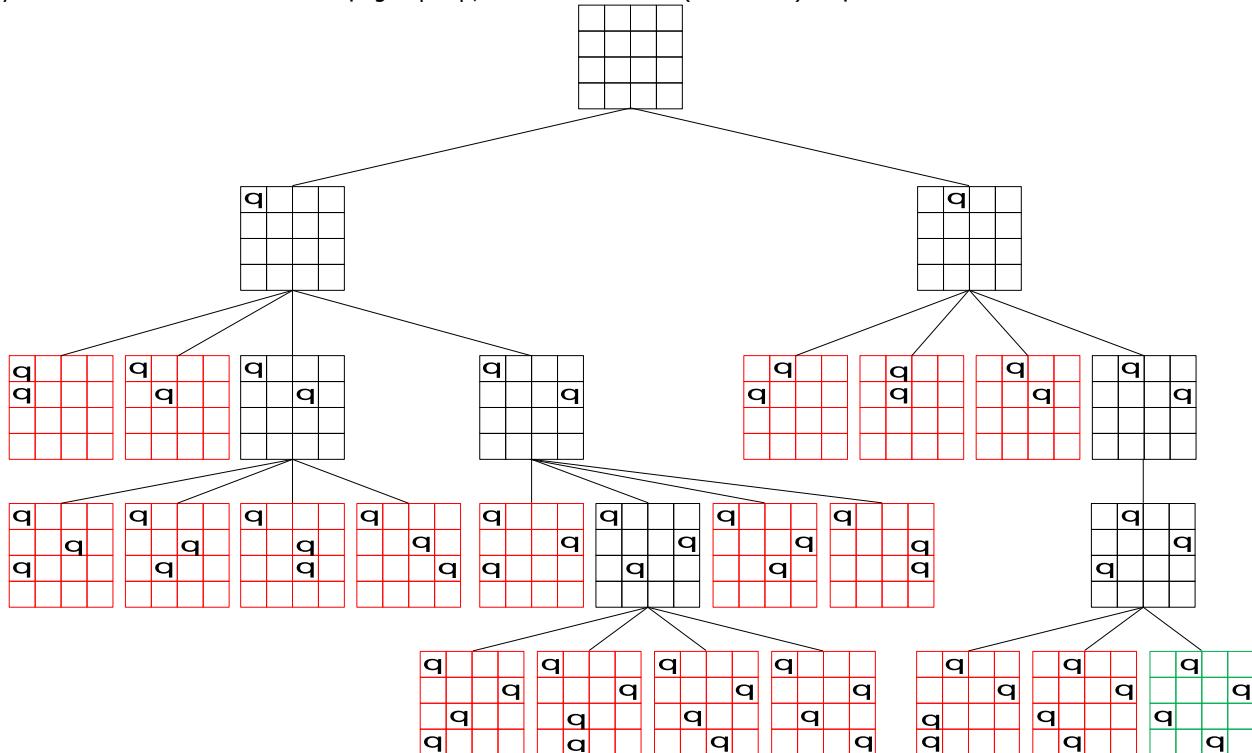


Backtracking là chiến lược tìm đến một mục tiêu bằng một chuỗi thao tác thử có hệ thống, với *khả năng quay lui* trên nhánh đang chọn để thử các nhánh khác. Khả năng quay lui này được cài đặt hiệu quả bằng kỹ thuật đệ quy.

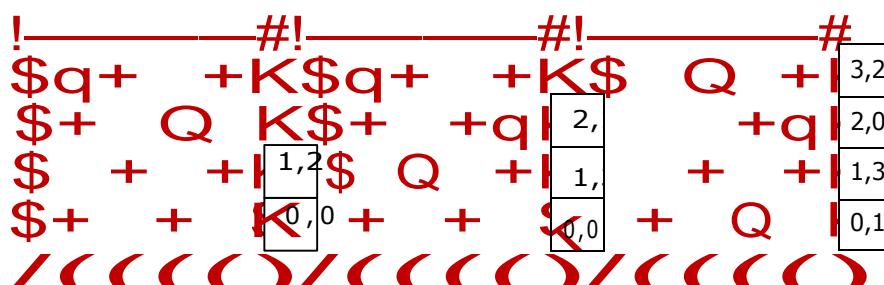
Bài toán áp dụng backtracking điển hình là bài toán tám quân Hậu:

Đặt tám quân Hậu trên bàn cờ vua kích thước 8×8 sao cho không có quân Hậu nào có thể tấn công được quân Hậu khác. Nói cách khác, lời giải của bài toán là cách xếp tám quân Hậu trên bàn cờ vua 8×8 sao cho không có hai quân Hậu nào đứng trên cùng hàng, cùng cột hoặc cùng đường chéo.

Duyệt không gian giải pháp giống như duyệt theo chiều sâu trên một cây quyết định. Tuy nhiên, hiếm khi phải lưu trữ toàn bộ cây, mà chỉ một đường dẫn tới một gốc được lưu trữ, để cho phép quay lui. Để đơn giản hóa, ta xét bài toán 4 Hậu. Hình dưới là cây tìm kiếm cho đến lúc tìm ra một giải pháp, các node "bế tắc" (dead-end) được tô màu đỏ.



Hình sau giải thích trường hợp quay lui, với run-time stack bên phải bàn cờ:



Trái: bắt đầu bằng cách đặt quân Hậu thứ nhất tại góc trên trái ($0,0$). Xét hàng 1, quân Hậu thứ hai được tại vị trí hợp lệ đầu tiên ($1,2$). Xét hàng 2, không còn vị trí nào hợp lệ.

Giữa: ta phải quay lui về thời điểm đặt quân Hậu thứ hai để chọn đường đi khác, đặt quân Hậu thứ hai tại vị trí ($1,3$). Xét hàng 2, quân Hậu thứ ba được tại vị trí hợp lệ đầu tiên ($2,1$). Xét hàng 3, không còn vị trí nào hợp lệ.

Phải: ta phải quay lui về thời điểm đặt quân Hậu thứ nhất để chọn đường đi khác, đặt quân Hậu thứ nhất tại vị trí (0,1). Xét hàng 1, đặt quân Hậu thứ hai tại vị trí hợp lệ (1,3). Xét hàng 2, đặt quân Hậu thứ ba tại vị trí hợp lệ (2,0). Xét hàng 3, đặt quân Hậu thứ tư tại vị trí hợp lệ (3,2). Ta đã tìm ra một giải pháp.

Cách tiếp cận bằng backtracking: giải pháp được thể hiện bởi vector (v_1, \dots, v_n) lưu các trị đã đi qua với cách duyệt theo chiều sâu, cho đến khi tìm thấy giải pháp. Khởi tạo từ một vector rỗng, tại mỗi bước vector giải pháp mở rộng thêm một trị mới. Nếu vector thành phần (v_1, \dots, v_i) không dẫn đến giải pháp, nghĩa là không thể hiện như một phần của giải pháp, quay lui được thực hiện bằng cách loại bỏ trị vi cuối vector thành phần, sau đó tiếp tục bằng cách thử mở rộng vector với một trị khác có thể.

```
import java.io.PrintStream;
public class Queens {
    final boolean available = true;
    int squares = 4, norm = squares - 1;
    int[] positionInRow = new int[squares];
    boolean[] column = new boolean[squares];
    boolean[] leftDiagonal = new boolean[squares * 2 - 1];
    boolean[] rightDiagonal = new boolean[squares * 2 - 1];
    int howMany = 0;

    public Queens() {
        for (int i = 0; i < squares; ++i) {
            positionInRow[i] = -1;
            column[i] = available;
        }
        for (int i = 0; i < squares * 2 - 1; ++i) {
            leftDiagonal[i] = rightDiagonal[i] = available;
        }
    }

    public void print(PrintStream out) {
        out.println("Solution #" + ++howMany);
        for (int i = 0; i < squares; ++i) {
            for (int j = 0; j < squares; ++j) {
                out.print(j == positionInRow[i] ? "[Q]" : "[ ]");
            }
            out.println();
        }
    }

    public void putQueen(int row) {
        for (int col = 0; col < squares; ++col) {
            if (column[col] == available
                && leftDiagonal[row + col] == available
                && rightDiagonal[row - col + norm] == available) {
                positionInRow[row] = col;
                column[col] = !available;
                leftDiagonal[row + col] = !available;
                rightDiagonal[row - col + norm] = !available;
                if (row < squares - 1) {
                    putQueen(row + 1);
                } else {
                    print(System.out);
                }
                column[col] = available;
                leftDiagonal[row + col] = available;
                rightDiagonal[row - col + norm] = available;
            }
        }
    }

    public static void main(String[] args) {
        new Queens().putQueen(0);
    }
}
```

Trong putQueen(row):

- Điều kiện đặt quân Hậu hợp lệ là các ô của cột (mảng column), đường chéo trái (mảng leftDiagonal) và đường chéo phải (mảng rightDiagonal) đi qua vị trí đang xét đều chưa đặt quân (available).
- Mảng positionInRow chính là vector giải pháp, chỉ số row của mảng chỉ định dòng đang xét, trị lưu tại positionInRow[row] là chỉ số cột, nơi thử đặt quân Hậu trên dòng row tương ứng. Vòng lặp for sẽ thử đặt quân Hậu trên tất cả các cột của dòng row.
- Mỗi bước của vòng lặp for duyệt một nhánh giải pháp, với gốc là vị trí thử đặt quân Hậu trên dòng row tương ứng. Nếu đặt được quân Hậu tại dòng row, putQueen() sẽ được gọi đệ quy để thử đặt quân Hậu trên dòng kế tiếp (row + 1). Nếu vị trí (col)

đặt quân Hậu tại dòng row không hợp lệ, vị trí tiếp theo (++col) trên dòng row sẽ được thử. Nếu việc gọi đệ quy dẫn đến mọi dòng đều đặt quân Hậu thành công (đệ quy cho đến khi row bằng $\text{square} - 1$), một giải pháp được tìm thấy và xuất kết quả.

Cây nhị phân tìm kiếm

Với danh sách liên kết, dù được sắp xếp, bạn vẫn phải tìm kiếm tuyến tính kể từ node đầu danh sách. Nếu tổ chức danh sách liên kết thành cây theo một điều kiện sắp xếp nào đó, số node phải duyệt khi tìm kiếm sẽ giảm đi rất nhiều. Một cây tìm kiếm nhị phân (BST - Binary Search Tree) có thể đáp ứng được yêu cầu trên.

Cây BST là:

- Cây nhị phân: mỗi node của cây chỉ có tối đa hai con, con trái và con phải. Con có thể không có, nghĩa là tham chiếu chỉ đến node con bằng **null**. Node không có con nào gọi là node lá.

- Với mỗi node X của cây, trị (hoặc khóa tìm kiếm) chứa tại các node trong cây con bên trái của X đều nhỏ hơn trị tại node X, trị chứa tại các node trong cây con bên phải của X đều lớn hơn trị tại node X.

Cấu trúc node của cây nhị phân:

```
public class BSTNode<E> {
    protected E info;
    protected BSTNode<E> left, BSTNode<E> right;

    public BSTNode(E info, BSTNode<E> left, BSTNode<E> right) {
        this.info = info;
        this.left = left;
        this.right = right;
    }

    public BSTNode(E info) {
        this(info, null, null);
    }
}
```

I. Thay đổi cây

Một số tác vụ có thể làm thay đổi cây một cách hệ thống: thêm node, xóa node, cập nhật dữ liệu của node, cân bằng cây.

```
public class BST<E extends Comparable<? super E>> {
    protected BSTNode<E> root = null;
}
```

1. Thêm node

Thêm một node vào cây nghĩa là cập nhật trường **left** hoặc **right** cho *node cha của node đó*. Vì thế, thao tác quan trọng khi thêm node là *tim node cha của node được thêm vào*. Trường hợp ngoại lệ, node thêm vào là node đầu tiên của cây nên không có node cha, ta cập nhật **root** chỉ đến node đó.

Để xác định node cha của node định thêm vào:

- Bắt đầu từ node gốc, so sánh trị định thêm vào với trị lưu tại node đang xét, do **p** chỉ đến. Nếu trị định thêm vào nhỏ hơn, chuyển **p** sang cây con bên trái rồi so sánh tiếp. Nếu trị định thêm vào lớn hơn, chuyển **p** sang cây con bên phải rồi so sánh tiếp. Ta dùng thủ thuật "hai con trỏ theo nhau": **prev** theo ngay sau **p**; khi **p** chỉ đến **null** thì **prev** chỉ node cha của **p**.

- Khi kết thúc vòng lặp tìm node cha, nếu:

- + **prev** là **null** thì cây đang rỗng, node thêm vào là node đầu tiên của cây, ta cập nhật **root** chỉ đến node mới.
- + **prev** khác **null**, ta phải xác định sẽ cập nhật trường nào (**left** hoặc **right**) của node cha **prev**. Ta thực hiện bằng cách so sánh trị định thêm vào với trị mà node cha **prev** đang chứa. Nếu trị định thêm vào nhỏ hơn, cập nhật **prev.left** chỉ đến node mới; nếu trị định thêm vào lớn hơn, cập nhật **prev.right** chỉ đến node mới.

```
public void insert(E info) {
    BSTNode<E> prev = null, p = root;
    while (p != null) {
        prev = p;
        if (info.compareTo(p.info) < 0) p = p.left;
        else p = p.right;
    }
    // prev ngay sau p, prev là null thì cây rỗng
    // ngược lại, prev là node lá và t là null
    if (prev == null) root = new BSTNode<>(info);
    else if (info.compareTo(prev.info) < 0)
        prev.left = new BSTNode<>(info);
    else
        prev.right = new BSTNode<>(info);
}
```

2. Xóa node

Các nguyên tắc:

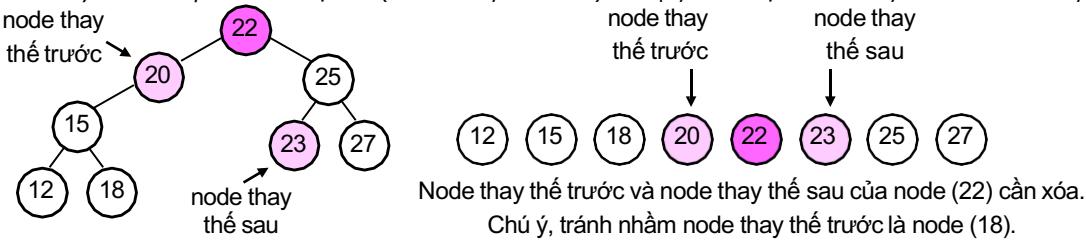
- Muốn xóa một node ta *phải cập nhật node cha của nó*, ta tìm được node cha này bằng thủ thuật "hai con trỏ theo nhau". Con trỏ **p** di chuyển để tìm node cần xóa, con trỏ **prev** di chuyển theo ngay sau **p**. Khi **p** chỉ đến node cần xóa, **prev** chỉ đến node cha của **p**. Trường hợp node cần xóa là node gốc **root** của cây (**p** bằng **root** và **prev** bằng **null**), ta phải chú ý cập nhật **root** khi xóa node.

- Độ phức tạp của giải thuật xóa một node phụ thuộc vào số cây con mà node đó có:

- + Node không có con (node lá): tham chiếu của node cha chỉ đến nó được đặt là **null** và node đó sẽ bị loại bỏ khỏi cây.

- + Node có một con: tham chiếu của node cha chỉ đến node cần xóa được đặt lại để *chỉ đến node con* của node cần xóa.
 - + Node có hai con: có hai giải pháp, xóa node bằng ghép cây (merging) và xóa node bằng sao chép (copying).
 - Cả hai giải pháp xóa node đều cần *tìm node thay thế*.
- Node thay thế là node có sẵn trong cây, khi thay thế vị trí của node bị xóa vẫn bảo đảm tính chất của cây BST. Nghĩa là: node thay thế phải có trị lớn hơn tất cả các trị chứa trong cây con bên trái của node bị xóa, hoặc phải có trị nhỏ hơn tất cả các trị chứa trong cây con bên phải node bị xóa.

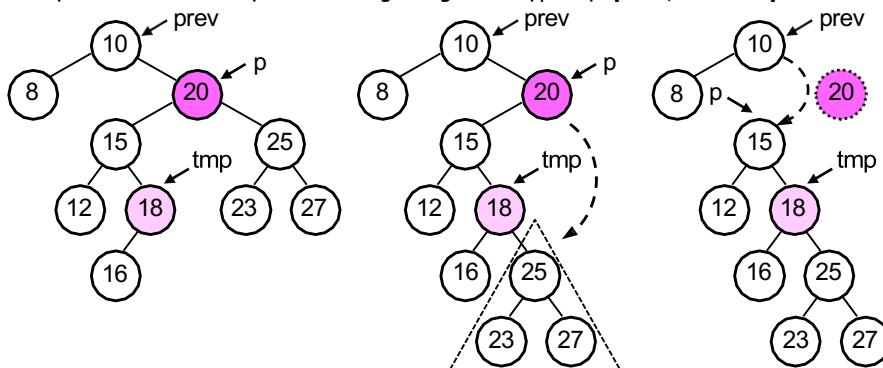
Để đáp ứng yêu cầu trên, node thay thế phải là node *cực phải của cây con bên trái* node bị xóa (node "thay thế trước") hoặc node *cực trái của cây con bên phải* node bị xóa (node "thay thế sau"). Ta quy ước chọn node thay thế là node "thay thế trước".



a) Xóa node bằng ghép cây

Giả sử node **p** là node cần xóa, **p** có hai con.

- Về sau ta sẽ xóa node **p**, vì vậy khi tìm **p**, dùng thủ thuật "hai con trỏ theo nhau" để biết node **prev** là cha của node **p**.
- Chuyển sang cây con bên trái của **p** và luôn đi về bên phải để đến node cực phải của cây con này, gọi là **tmp**. **Tmp** chỉ node "thay thế trước" của **p**.
- Cây con bên phải của node cần xóa **p** sẽ được "trao" lại cho node "thay thế trước" **tmp** quản lý. Cụ thể, cây con bên phải của **p** được ghép thành cây con bên phải của **tmp**: **tmp.right = p.right**;
- Bây giờ node **p** chỉ còn một con và có thể loại bỏ dễ dàng bằng cách cập nhật **prev**, cha của **p**.



- Khi cập nhật **prev** sau khi xóa node, chú ý trường hợp xóa node gốc **root**, phải cập nhật **root**.

```
public void deleteByMerging(E info) {
    BSTNode<E> tmp, t, p = root, prev = p;
    while (p != null && !p.info.equals(info)) {           // tìm trong cây node p chứa trị info cần xóa
        prev = p;
        p = (p.info.compareTo(info) < 0) ? p.right : p.left;
    }
    if (p != null && p.info.equals(info)) {
        t = p;
        if (t.right == null)                                // node không có con phải: node cha chỉ con trái của node cần xóa (1)
            t = t.left;
        else if (t.left == null)                            // node không có con trái: node cha chỉ con phải của node cần xóa (2)
            t = t.right;
        else {                                              // nếu node cần xóa có hai con (3)
            tmp = t.left;                                  // + chuyển sang cây con trái
            while (tmp.right != null)                      // + đi về cực phải để tìm node thay thế
                tmp = tmp.right;
            tmp.right = t.right;                          // kết thúc vòng lặp, tmp chỉ node thay thế
            t = t.left;                                  // con phải của node bị xóa thành con phải của node thay thế
        }
        // cập nhật cha của node bị xóa (hoặc root) với node t
        if (p == root) root = t;                         // nếu node cần xóa là root thì cập nhật root với node t
        else if (prev.left == p)                         // ngược lại, cập nhật cha của node cần xóa với node t
            prev.left = t;
        else prev.right = t;
    } else if (root != null) System.out.println("Node not found!");
    else System.out.println("Tree is empty");
}
```

Cách xóa node bằng ghép cây dễ làm cho cây mất cân bằng, gây ảnh hưởng đến hiệu quả tìm kiếm.

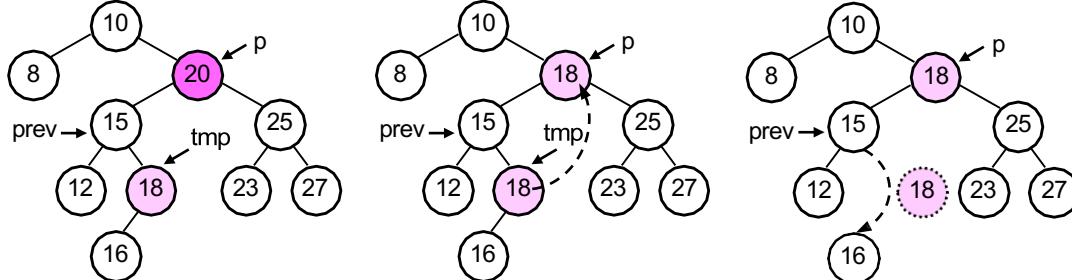
b) Xóa node bằng sao chép

Giả sử node **p** là node cần xóa, **p** có hai con.

- Chuyển sang cây con bên trái của **p** và luôn đi về bên phải để đi đến node cực phải của cây con này, gọi là **tmp**. Về sau ta sẽ xóa node **tmp**, vì vậy khi tìm node **tmp**, dùng thủ thuật "hai con trỏ theo nhau" để biết node **prev** là node cha của node **tmp**.
- **tmp** là node "thay thế trước" của **p**, nghĩa là nó có thể thay thế được **p**. Sao chép chồng dữ liệu chứa trong node "thay thế trước" **tmp** vào node cần xóa **p**.
- Bây giờ có hai node chứa dữ liệu giống nhau: **p** và **tmp**, ta sẽ xóa node **tmp**; vì node này hoặc là node lá, hoặc chỉ có một cây con bên trái nên dễ dàng loại bỏ hơn.

Việc xóa node **tmp** có ngoại lệ. Thông thường, với **prev** là node cha của **tmp**, do **tmp** là node cực phải nên **tmp** thường là con phải của **prev**, cập nhật khi xóa: **prev.right = tmp.left**. Tuy nhiên, nếu con trái của node bị xóa **p** không có con phải, con trái của **p** sẽ là node **tmp** và khi đó **tmp** cũng là con trái của **prev**, cập nhật khi xóa: **prev.left = tmp.left**. Trường hợp đặc biệt này được nhận biết khi **prev = p**.

- Cập nhật sau khi xóa node.



```

public void deleteByCopying(E info) {
    BSTNode<E> t, p = root, previous = p;
    while (p != null && !p.info.equals(info)) {           // tìm trong cây node p chứa trị info cần xóa
        previous = p;          // previous là cha của p, dùng khi xóa p với p chỉ có 1 con
        p = (p.info.compareTo(info) < 0) ? p.right : p.left;
    }
    if (p != null && p.info.equals(info)) {
        t = p;
        if (t.right == null)           // node không có con phải: node cha chỉ con trái của node bị xóa (1)
            t = t.left;
        else if (t.left == null)       // node không có con trái: node cha chỉ con phải của node bị xóa (2)
            t = t.right;
        else {                         // nếu node cần xóa có hai con (3)
            BSTNode<E> tmp = t.left; //     + chuyển sang cây con trái
            BSTNode<E> prev = t;    //     + tmp di trước, prev theo ngay sau
            while (tmp.right != null) { //     + đi về cực phải để tìm node thay thế
                prev = tmp;
                tmp = tmp.right;
            }
            t.info = tmp.info;        //     kết thúc vòng lặp, tmp chỉ node thay thế
            if (prev == t)            //     sao chép chồng trị của node thay thế lên node bị xóa
                prev.left = tmp.left; //     trường hợp đặc biệt, cập nhật prev.left
            else prev.right = tmp.left; //     trường hợp bình thường, cập nhật prev.right
        }
        // cập nhật cha của node bị xóa (hoặc root) với node t
        if (p == root) root = t;      // nếu node bị xóa là root thì cập nhật root với node t
        else if (previous.left == p)  // ngược lại, cập nhật cha của node cần xóa với node t
            previous.left = t;
        else previous.right = t;
    } else if (root != null) System.out.println("Node not found!");
    else System.out.println("Tree is empty");
}

```

II. Duyệt cây

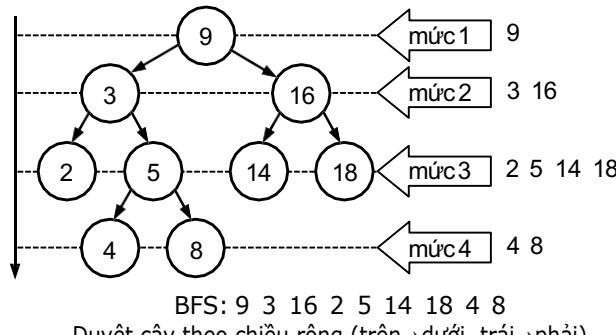
Duyệt cây là quá trình thăm mỗi node trong cây đúng một lần. Có nhiều cách duyệt cây, nhưng ta chỉ quan tâm đến các cách duyệt có tính sử dụng cao:

- Duyệt theo chiều rộng (breadth-first traversal)
 - Duyệt theo chiều sâu (depth-first traversal)
- Mỗi cách duyệt cây phục vụ cho một mục đích khác nhau.

1. Duyệt theo chiều rộng

Duyệt theo chiều rộng còn gọi là duyệt theo mức. Cách duyệt phổ biến nhất là "từ trên xuống dưới, từ trái sang phải": lần lượt duyệt từng mức (level-by-level) của cây từ trên xuống dưới, tại mỗi mức duyệt các node từ trái sang phải.

Duyệt theo chiều rộng thường dùng với cây quyết định. Ví dụ trong chương trình chơi cờ, tại nước đang xét ta cần duyệt để đánh giá tất cả nước đi kế tiếp có thể, tức duyệt các node tại mức kế tiếp node đang xét.



Cài đặt duyệt theo chiều rộng được thực hiện bằng queue:

- node được duyệt lấy từ đầu queue. Vì vậy, khi bắt đầu duyệt phải đặt node gốc vào queue rồi mới lấy ra duyệt.
- sau khi duyệt một node lấy ra từ queue, con trái rồi con phải của nó (nếu có) được đặt vào cuối queue.

Điều này bảo đảm duyệt hết các node tại một mức rồi mới duyệt đến các con của nó ở mức kế tiếp.

```
public void BFS() {
    BSTNode<E> p = root;
    Queue<BSTNode<E>> queue = new LinkedList<>();
    if (p != null) {
        queue.add(p); // đặt root vào trong queue
        while (!queue.isEmpty()) {
            p = queue.remove();
            visit(p);
            if (p.left != null) queue.add(p.left);
            if (p.right != null) queue.add(p.right);
        }
    }
}
```

Duyệt theo mức là cơ sở biểu diễn cây bằng mảng, đặc biệt thuận lợi với cây cân bằng hoàn hảo.

2. Duyệt theo chiều sâu

Trên cây ta đi xa nhất về phía bên trái (hoặc phải), sau đó *quay ngược trở lại* (backtracking) cho đến khi gặp chỗ rẽ (node cha) đầu tiên, chuyển sang cây con phải (hoặc trái) rồi tiếp tục đi xa nhất về bên trái (hoặc phải) nếu có thể. Lặp lại cho đến khi các node đều được thăm.

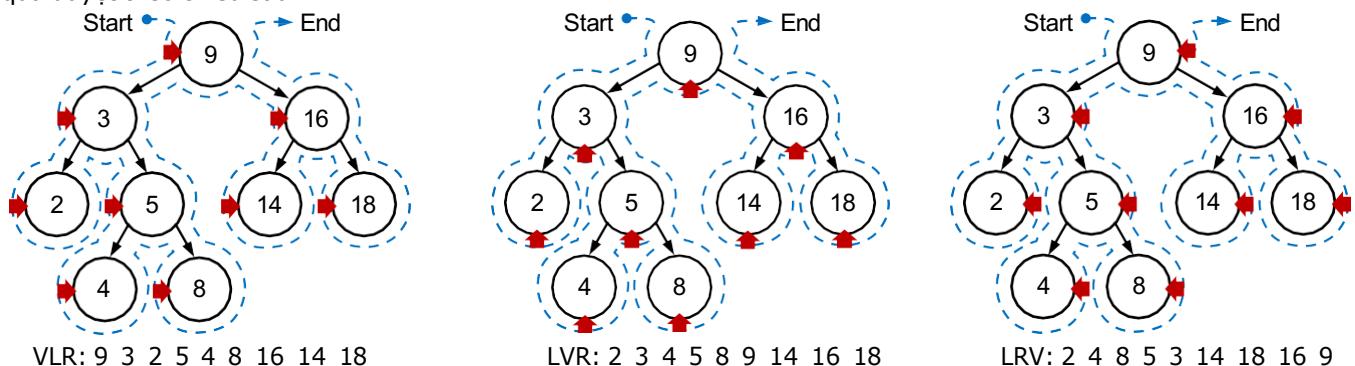
Do quá trình trên không chỉ rõ node được thăm khi nào, nên có nhiều cách duyệt theo chiều sâu, thường ta chỉ quan tâm đến ba cách khi duyệt *từ trái sang phải*, khác nhau ở thứ tự thăm node:

VLR: duyệt cây tiền thứ tự (preorder), thăm node - duyệt VLR cây con trái - duyệt VLR cây con phải.

LVR: duyệt cây trung thứ tự (inorder), duyệt LVR cây con trái - thăm node - duyệt LVR cây con phải.

LRV: duyệt cây hậu thứ tự (postorder), duyệt LRV cây con trái - duyệt LRV cây con phải - thăm node.

Từ đặc điểm duyệt chiều sâu trên, ta có phương pháp duyệt cây một cách thủ công, giúp kiểm tra nhanh và chính xác các kết quả duyệt theo chiều sâu:



- Đi theo một "đường bao" xung quanh các node cây từ trái sang phải (nếu trong thứ tự duyệt L trước R).

Khi di chuyển ta duyệt qua các node của cây 3 lần, tương ứng với ba cách duyệt cây.

- Thứ tự xuất trị của các node (thăm node) được xác định như sau (chú ý các mũi tên)

⇒ VLR: (preorder) xuất trị của node khi đi qua ngay "trước" (bên trái) node.

LVR: (inorder) xuất trị của node khi đi qua ngay "dưới" node.

LRV: (postorder) xuất trị của node khi đi qua ngay "sau" (bên phải) node.

Nhận xét:

- Duyệt LVR một cây BST cho kết quả là các node với trị tăng dần.

- Từ kết quả duyệt VLR hoặc LRV của cây BST, bạn có thể tạo lại cây. Ví dụ, cho kết quả duyệt VLR: 9 3 2 5 4 8 16 14 18. Ta có node gốc của cây là node đầu tiên: 9, cây con bên trái chứa các node có trị < 9: 3 2 5 4 8, cây con bên phải chứa các node có trị > 9: 16 14 18. Node gốc của cây con bên trái là node đầu tiên trong dãy: 3. Node gốc của cây con bên phải là node đầu tiên trong dãy: 16. Suy luận tương tự để tạo lại cây.

Cây là một cấu trúc dữ liệu có bản chất đệ quy nên duyệt theo chiều sâu được cài đặt dễ dàng bằng đệ quy (dùng run-time stack), hoặc cài đặt không đệ quy bằng cách dùng stack tường minh. Stack được dùng trong cài đặt thao tác duyệt với mục đích quay ngược trở lại (backtracking) node cha khi duyệt.

Duyệt VLR đệ quy dựa vào cấu trúc đệ quy của cây:

```
public void VLR(BSTNode<E> p) {
    if (p != null) {
        visit(p);
        VLR(p.left);
        VLR(p.right);
    }
}
```

Duyệt VLR không đệ quy, so sánh với duyệt theo chiều sâu ta thấy cách duyệt này dùng stack để có thể quay lui. Ngoài ra, để duyệt trái (L) trước phải (R), con phải được đưa vào stack trước con trái.

```
public void VLR() {
    BSTNode<E> p = root;
    Stack<BSTNode<E>> stack = new Stack<>();
    if (p != null) {
        stack.push(p);
        while (!stack.isEmpty()) {
            p = stack.pop();
            visit(p);
            if (p.right != null) stack.push(p.right);
            if (p.left != null) stack.push(p.left);
        }
    }
}
```

3. Duyệt theo chiều sâu không dùng stack

Hai giải thuật đặc biệt được dùng để duyệt cây theo chiều sâu không dùng stack là:

- Dùng cây threaded.

- Duyệt cây bằng phép biến đổi cây của J. Morris.

Phần này chỉ xem xét duyệt inorder. Với hiệu chỉnh thích hợp, có thể dùng hai giải thuật trên để duyệt preorder và postorder.

a) Cây threaded

Thread là khái niệm mở rộng của liên kết trái và phải của một node:

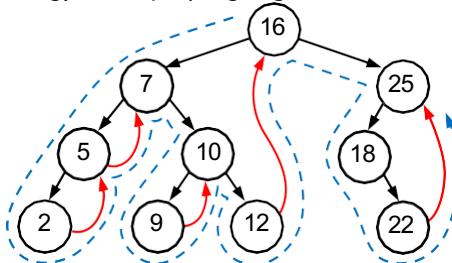
- Liên kết trái chỉ đến con trái, hoặc chỉ đến node predecessor; là node có thứ tự *ngay trước* node đang xét khi duyệt inorder.

- Liên kết phải chỉ đến con phải, hoặc chỉ đến node successor; là node có thứ tự *ngay sau* node đang xét khi duyệt inorder.

Dễ nhận thấy, *node trái nhất* trong cây có liên kết trái là **null** vì không có predecessor; *node phải nhất* trong cây có liên kết phải là **null** vì không có successor.

Cây có các node dùng thread như trên gọi là cây threaded.

Thực tế chỉ cần một loại thread, ví dụ duyệt từ trái qua phải ta chỉ cần thread phải (chỉ đến successor) như hình dưới. Thread phải cho phép quay ngược trở lại (backtracking) nên hiệu quả giống như đi trên "đường bao" của cây:



Thread phải (mũi tên màu đỏ) hỗ trợ cho duyệt cây theo chiều sâu
Node (25) có liên kết phải là **null**. Đường không liên tục là "đường bao"

Cấu trúc node của cây threaded:

```
public class TTNode<E> {
    protected TTNode<E> left, right, thread;
    protected E info;

    public TTNode(TTNode<E> left, E info, TTNode<E> right, TTNode<E> thread) {
        this.left = left;
        this.info = info;
        this.right = right;
        this.thread = thread;
    }

    public TTNode(E info) {
        this(null, info, null, null);
    }
}
```

Cây threaded:

```
public class ThreadedTree<E extends Comparable<? super E>> {
    protected TTNode<E> root = null;
}
```

Thao tác chèn node mới vào cây threaded:

```
public void insert(E info) {
    TTNode<E> t = new TTNode<>(info);
    if (root == null) {
        root = t;
        return;
    }
    TTNode<E> p = root, prev = p;
    while (p != null) {
        prev = p;
        if (info.compareTo(p.info) < 0) p = p.left;
        else if (p.thread == null) p = p.right;
        else break;
    }
    if (info.compareTo(prev.info) < 0) {
        prev.left = t;
        t.thread = prev;
    } else if (prev.thread != null) {
        t.thread = prev.thread;
        prev.thread = null;
        prev.right = t;
    } else {
        prev.right = t;
    }
}
```

Trong duyệt cây threaded, khi xét con phải ta cần phân biệt hai trường hợp: liên kết phải và thread phải.

Giải thuật

Gọi node **p** là node cần xét. **p** bắt đầu từ **root**.

1. Tìm đến node trái nhất. Đến bước 2.
2. Thăm node. Đến bước 3.
3. Nếu node đang xét có con phải, đến bước 4; ngược lại, tức có thread phải, đến bước 5.
4. Chuyển sang node con phải, tìm node trái nhất của cây con phải. Trở lại bước 2.
5. Chuyển sang node do thread chỉ đến, trở lại bước 2. (backtracking)
6. Thực hiện vòng lặp từ 2 đến 5 cho đến khi **p** bằng **null**. Lúc này **p** là liên kết phải, có trị **null** duy nhất trong cây.

```
public void inorderThread() {
    TTNode<E> p = root;
    if (p != null) {
        while (p.left != null) p = p.left;
        while (p != null) {
            visit(p);
            if (p.thread != null) p = p.thread;
            else {
                p = p.right;
                if (p != null)
                    while (p.left != null) p = p.left;
            }
        }
    }
}
```

b) Duyệt cây bằng phép biến đổi cây của J. Morris

Ý tưởng là bổ sung các liên kết tạm để biến đổi từng phần của cây thành không có con trái. Cây trở thành backbone nên thuận lợi cho việc duyệt tuyến tính. Tuy vậy, các liên kết ban đầu của cây vẫn được giữ nguyên. Sau khi thăm node, liên kết tạm không còn cần thiết, cắt bỏ liên kết tạm sẽ đưa cây trở về cấu trúc ban đầu.

Định nghĩa **tmp**: với mỗi **p** có con trái, **tmp** là *node phải nhất của cây con bên trái p, không vượt quá p*. Chú ý, **tmp** là node thay thế trước của **p**. Tìm **tmp** bằng cách qua con trái rồi luôn đi về phải nhưng không vượt quá **p**.

```
tmp = p.left;
while (tmp.right != null && tmp.right != p)
    tmp = tmp.right;
```

Xét **p** bắt đầu từ **root**, với mỗi **p** có 2 trường hợp cần xử lý:

- nếu **p** không có con trái: thăm **p** rồi duyệt tuyến tính sang phải (**p = p.right**).

- nếu **p** có con trái, ta tìm **tmp** như đã trình bày ở trên, sau đó xét **tmp**:

+ **tmp.right = null**: tạo liên kết tạm chỉ đến **p** (**tmp.right = p**), chuyển sang xét con trái (**p = p.left**).

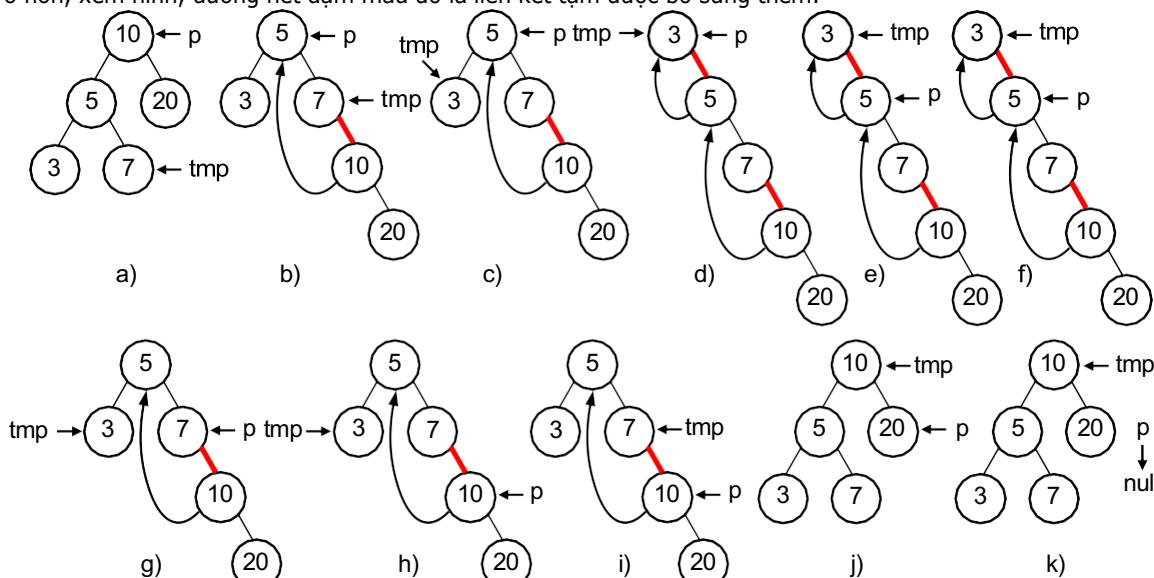
+ **tmp.right ≠ null**: **tmp.right** chính là liên kết tạm, cắt liên kết tạm này (**tmp.right = null**), thăm **p**, chuyển sang xét con phải (**p = p.right**).

Điểm chính trong duyệt cây theo Morris, là tạo liên kết tạm (khi `tmp.right = null`) để duyệt cây tuyến tính và cắt liên kết tạm (khi `tmp.right ≠ null`) để trả cây trở lại cấu trúc ban đầu.

Cài đặt:

```
public void inorderMorris() {
    BSTNode<E> p = root, tmp;
    while (p != null) {
        if (p.left == null) {
            visit(p);
            p = p.right;
        } else {
            tmp = p.left;
            while (tmp.right != null && tmp.right != p)
                tmp = tmp.right; // tìm node phải nhất trong cây trái của p, không vượt quá p
            if (tmp.right == null) { // tạo liên kết tạm tmp.right
                tmp.right = p;
                p = p.left;
            } else {
                visit(p); // thăm node, sau đó ...
                tmp.right = null; // cắt liên kết tạm để phục hồi cây
                p = p.right;
            }
        }
    }
}
```

Để hiểu rõ hơn, xem hình, đường nét đậm màu đỏ là liên kết tạm được bổ sung thêm:



a) Xét `p` có con trái, tìm `tmp` là node phải nhất của cây con trái, `tmp.right = null` nên tạo liên kết tạm: `tmp.right = p`, xét tiếp con trái: `p = p.left`. Ta có b).

i) Xét `p` có con trái, tìm `tmp` là node phải nhất của cây con trái (node 7, không vượt quá `p`), `tmp.right ≠ null` nên ta thăm `p`, rồi cắt liên kết tạm: `tmp.right = null` làm nhánh trái của `p` hồi phục, sang phải xét tiếp: `p = p.right`. ta có j).

III. Cân bằng cây

Hiệu quả tìm kiếm trong cây phu thuộc rất nhiều vào hình dạng của cây. Nếu các node phân bố không cân bằng, chiều cao của cây không tối ưu, hiệu quả tìm kiếm sẽ giảm rõ rệt. Nếu mỗi mức chỉ có một node, mỗi node chỉ có con phải ta có *cây backbone*; mỗi node có con trái và con phải xen kẽ ta có *cây zigzag*. Cây backbone và cây zigzag bản chất là danh sách liên kết nên tìm kiếm trở thành tuyến tính.

Một cây được gọi là *cân bằng* nếu chiều cao của hai cây con của một node bất kỳ chỉ chênh lệch nhau một mức.

Một cây *hoàn hảo* (perfect tree), còn gọi là cây đầy (full tree), mọi node đều có hai con, trừ những node ở mức cao nhất đều là node lá. Một cây *cân bằng hoàn hảo* (perfectly balanced tree) nếu nó là cây hoàn hảo ngoại trừ mức thấp nhất, tại mức thấp nhất các node lá nằm hết về bên trái. Cây cân bằng hoàn hảo còn gọi là cây đầy đủ trái.

Trong một cây nhị phân có gốc tại mức 1, số node thuộc mức k đầy đủ là: 2^{k-1} và số node trong cây nhị phân đầy đủ đến mức k là: $2^k - 1$. Như vậy, cây cân bằng hoàn hảo lưu n node phải có chiều cao $\lceil \lg(n + 1) \rceil$; nói cách khác là phải kiểm tra $\lceil \lg(n + 1) \rceil$ node để tìm một phần tử. $\lceil x \rceil$ là trị nguyên lớn hơn x gần x nhất, tức trị làm tròn lên của x (ceiling).

Có một số cách để cân bằng cây: sắp xếp các phần tử để xây dựng lại cây hoặc cân bằng lại cây khi thêm/xóa phần tử.

1. Sắp xếp các phần tử để xây dựng lại cây

Dữ liệu các node từ cây được lưu ra một mảng và được sắp xếp tăng trên mảng bằng một giải thuật sắp xếp nào đó, thường ta duyệt inorder và đưa các node duyệt được vào mảng. Sau đó trị giữa của mảng được chọn để chèn vào làm node gốc của cây. Như vậy mảng chia làm hai mảng con trái và phải, tương ứng với dữ liệu của cây con trái và cây con phải của node vừa chèn. Tiến hành chọn nhị phân tương tự như vậy với các mảng con.

Dễ dàng cài đặt giải thuật trên bằng đệ quy:

```
void balance(ArrayList<E> a, int first, int last) {
    if (first <= last) {
        int middle = (first + last) / 2;
        insert(a.get(middle));
        balance(a, first, middle - 1);
        balance(a, middle + 1, last);
    }
}
```

2. Cấu trúc lại cây khi thêm/xóa phần tử

a) Giải thuật DSW (Colin Day, Quentin F. Stout, Bette L. Warren)

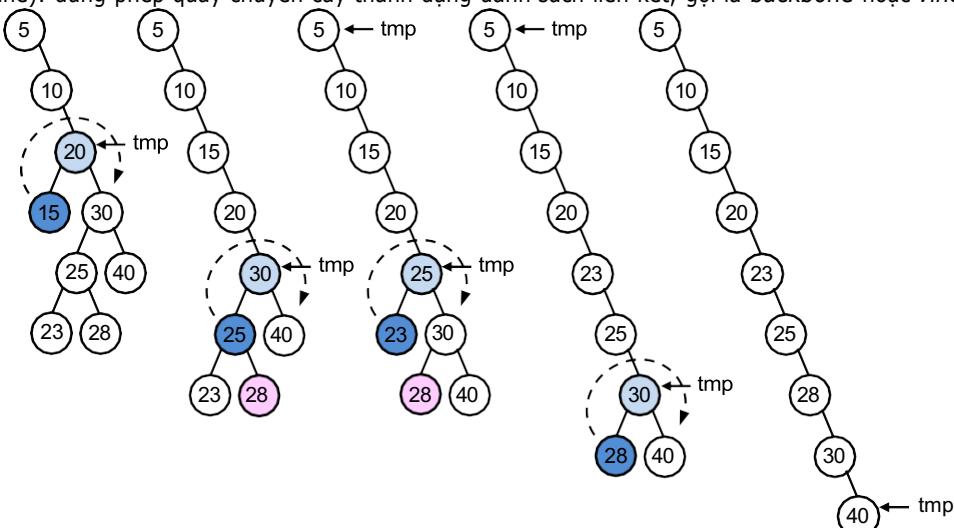
Trung tâm của giải thuật này là phép quay cây. Có hai phép quay: quay trái và quay phải; do chúng đối xứng nên chỉ cần mô tả phép quay phải. Giải thuật quay phải node Ch (node con, child) quanh node Pr (node cha, parent) như sau:

rotateRight(Gr, Pr, Ch)

```
if Pr không là node root của cây           // tức Gr không null
    node Gr trở thành node cha của Ch thay cho Pr;
    con phải của Ch trở thành con trái của Pr;
    node Ch nhận Pr làm con phải;
```

Giải thuật DSW gồm hai bước:

- Bước 1 (tree-to-vine): dùng phép quay chuyển cây thành dạng danh sách liên kết, gọi là *backbone* hoặc *vine*.

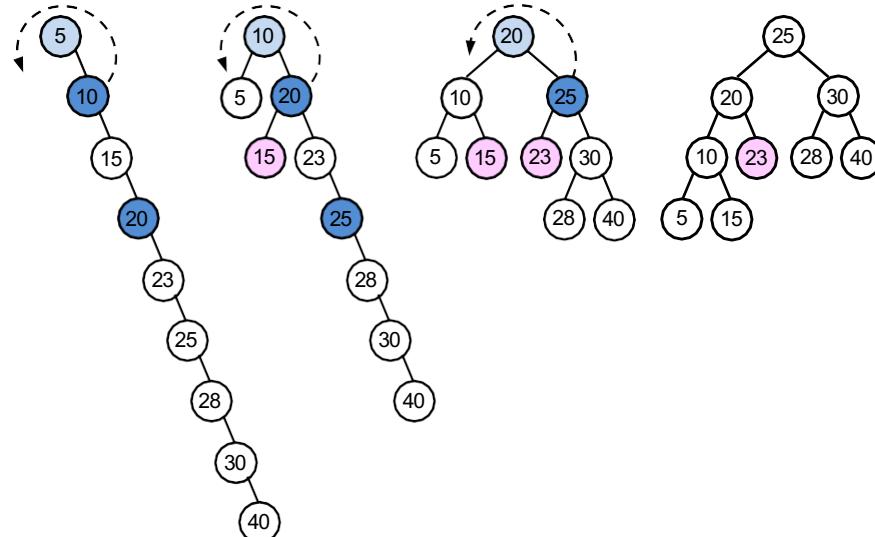


createBackbone(root, n)

```
tmp = root;
while (tmp != null)
    if tmp có con trái;
        quay con trái của tmp quanh tmp      // con trái thành cha của tmp;
        đặt tmp chỉ đến con trái (vừa thành cha mới);
    else đặt tmp chỉ đến con phải (duyệt tiếp);
```

Trong trường hợp xấu nhất, root không có con phải, vòng lặp thực hiện $2n - 1$ lần với $n - 1$ phép quay. Thời gian thực hiện của bước 1 là $O(n)$.

- Bước 2 (vine-to-tree): quay trái từng phần backbone để tạo cây cân bằng hoàn hảo.



Đi xuống trên backbone, quay trái từng node thứ hai quanh node cha của nó.

createPerfecTree(n)

```
m = 2lg(n + 1) - 1; // n là số node trong cây
thực hiện n - m phép quay bắt đầu từ đỉnh của backbone;
while (m > 1)
    m = m/2;
    thực hiện m phép quay bắt đầu từ đỉnh của backbone;
```

Với $\lfloor x \rfloor$ là trị nguyên nhỏ hơn x gần x nhất (làm tròn xuống, hàm floor).

Số lần quay cho bởi công thức: $n - m + (m - \lfloor \lg(m + 1) \rfloor) = n - \lfloor \lg(m + 1) \rfloor$. Như vậy số lần quay là $O(n)$.

b) Cây AVL (Adel'son - Vel'skii - Landis)

Cây AVL là một cây nhị phân mà chiều cao của cây trái và chiều cao của cây phải chênh lệch không quá 1. Như vậy, cây AVL là cân bằng, nhưng không cân bằng hoàn hảo. Các node của cây AVL có một hệ số cân bằng (balance factor) là hiệu của chiều cao của cây con phải và cây con trái, theo định nghĩa phải là $-1, 0, +1$.

Từ định nghĩa cây AVL, số node tối thiểu của một cây với chiều cao h được xác định bằng công thức truy hồi:

$$\min_h = \min_{h-1} + \min_{h-2} + 1, \text{ với } h > 1, \min_0 = 0, \min_1 = 1.$$

Công thức này (giống Fibonacci) suy ra giới hạn chiều cao h của cây AVL theo số lượng node n : $\lg(n + 1) \leq h \leq 1.44\lg(n + 2) - 0.328$

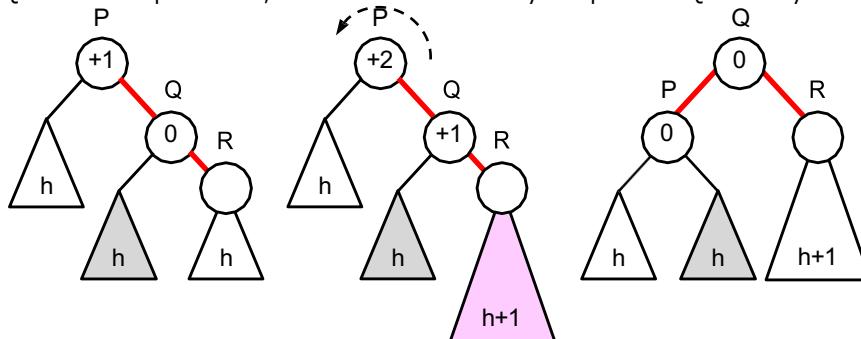
Do h giới hạn bởi $O(\lg n)$, trường hợp tìm kiếm xấu nhất cần $O(\lg n)$ phép so sánh. Với cây cân bằng hoàn hảo, ta có $h = \lceil \lg(n + 1) \rceil$. Như vậy, thời gian tìm kiếm trong trường hợp xấu nhất trong một cây AVL lớn hơn 44% so với trường hợp xấu nhất trên cây cân bằng hoàn hảo.

Nếu hệ số cân bằng của một node bất kỳ trong cây AVL bị vi phạm (nhỏ hơn -1 hoặc lớn hơn 1), thao tác cân bằng cây sẽ được thực hiện. Cây AVL mất cân bằng trong 4 trường hợp:

- + **RR** - cây con bên phải I có nhánh phải I dài
- + **RL** - cây con bên phải I có nhánh trái (L) dài
- + **LL** - cây con bên trái (L) có nhánh trái (L) dài
- + **LR** - cây con bên trái (L) có nhánh phải I dài

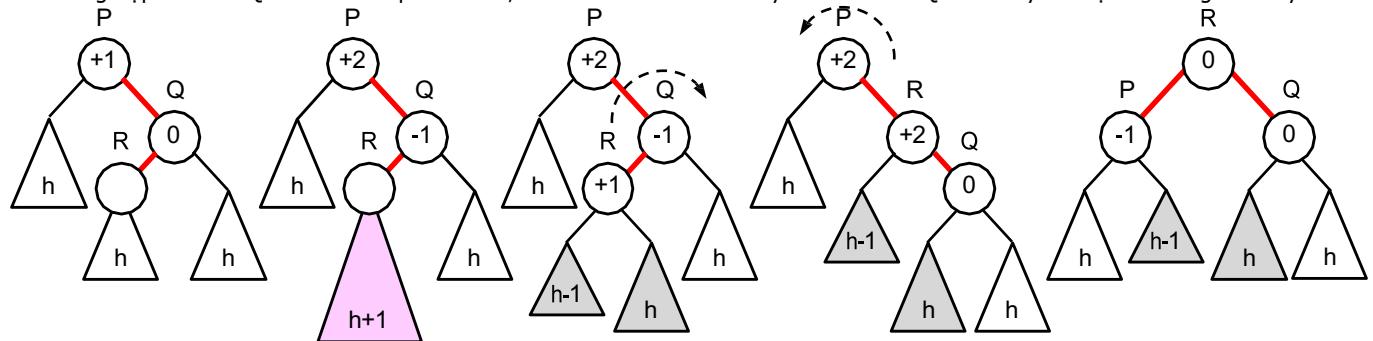
Do đối xứng, ta chỉ cần xem xét 2 trường hợp: RR và RL.

+ Trường hợp **RR**: với Q là node con phải của P, chèn thêm node vào cây con phải của Q làm thay đổi sự cân bằng của cây.



Thao tác tái bố trí là quay trái node Q quanh node P.

+ Trường hợp **RL**: với Q là node con phải của P, chèn thêm node vào cây con trái của Q làm thay đổi sự cân bằng của cây.

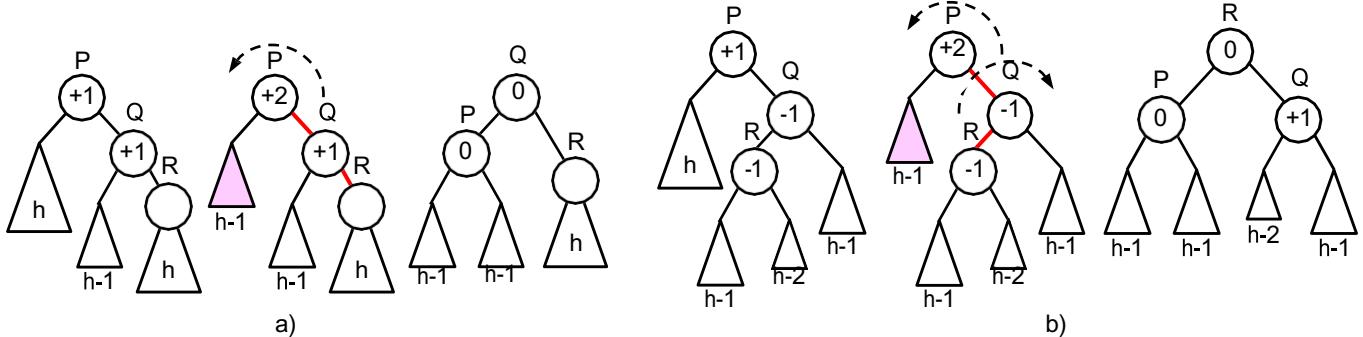


Thao tác tái bố trí thực hiện **quay kép**: quay phải cây con gốc Q (R quanh Q) để đưa về trường hợp RR, sau đó quay trái cây gốc P (Q quanh P) để giải quyết trường hợp RR.

P (node có hệ số cân bằng ± 2) có thể là cây riêng lẻ hoặc là cây con của một cây AVL lớn hơn. Việc mất cân bằng và khôi phục cân bằng của P không ảnh hưởng đến các node cha của P, vì sau khi cân bằng, cây P vẫn giữ nguyên chiều cao.

Do đó, khi chèn node gây mất cân bằng, ta cần tìm P tương ứng. P có thể tìm được bằng cách đi ngược từ node mới được chèn vào lên đến gốc của cây và cập nhật lại các hệ số cân bằng. Node đầu tiên ta gặp có hệ số cân bằng ± 1 bị thay đổi ± 2 sẽ là node gốc P của cây con phải cân bằng lại. Chú ý là hệ số cân bằng của node cha của P *không cần thay đổi*. Nếu không gặp node P như yêu cầu, ta chỉ cập nhật lại hệ số cân bằng của các node dọc theo nhánh đó mà không cần tái bố trí.

Khi xóa một node trên cây AVL, hệ số cân bằng được cập nhật lại từ node cha của node bị xóa cho đến node gốc. Trên nhánh này, nếu gặp node có hệ số cân bằng thay đổi thành ± 2 , thực hiện các phép quay cần thiết (tùy theo 4 trường hợp trên) để cân bằng lại cây con có node gốc là node đó. Chú ý là cần thực hiện cân bằng lại cho *tất cả* các node có hệ số cân bằng ± 2 trên nhánh của node bị xóa, *không dừng lại* sau node ± 2 đầu tiên như khi chèn. Trong trường hợp xấu nhất, mọi node trên nhánh bị xóa đều phải cân bằng nên mất $O(\lg n)$ phép quay. Đôi khi, sau khi xóa node không cần phép quay do cân bằng được cải thiện.



Một số trường hợp xóa node (cây màu hồng) và cân bằng lại cây, xoay một lần hoặc hai lần.

Các phân tích cho thấy, thêm và xóa node cần nhiều nhất $1.44\lg(n + 2)$ phép tìm kiếm. Thao tác thêm node cần thêm một hoặc hai phép quay. Thao tác xóa node cần thêm $1.44\lg(n + 2)$ trong trường hợp xấu nhất.

Bạn nên tự thử nghiệm việc chèn và xóa node trên cây AVL tại: <https://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html> để quan sát trực quan các trường hợp tái cân bằng cho cây.

Thực nghiệm cho thấy sau khi chèn node, 53% trường hợp không cần thêm thao tác cân bằng cây; khi xóa node (bằng phương pháp sao chép), 78% trường hợp không cần thêm thao tác cân bằng cây.

Cây AVL có thể mở rộng, cho phép chênh lệch chiều cao $\Delta > 1$, kết quả là chiều cao tăng theo Δ :

$$h = 1.81 \lg(n) - 0.71 \text{ nếu } \Delta = 2$$

$$h = 2.15 \lg(n) - 1.13 \text{ nếu } \Delta = 3$$

Kết quả thực nghiệm cho thấy, số node trung bình phải thăm khi tìm kiếm tăng khoảng một nửa so với cây AVL chuẩn ($\Delta = 1$) nhưng số thao tác cần hiệu chỉnh giảm đi rất nhiều.

IV. Heap

Heap là một cây nhị phân đặc biệt có các tính chất sau:

- Giá trị của mỗi node trong heap *không nhỏ hơn* giá trị lưu trong node con của nó.

- Cây *cân bằng hoàn hảo*, các lá ở bậc cuối cùng đều nằm đầy đủ các vị trí từ bên trái sang.

Tính chất thứ nhất định nghĩa một max heap. Tính chất thứ hai cho thấy số mức trong cây là $O(\lg n)$.

Để hiểu những tác vụ trên heap ta dùng cây nhị phân, nhưng khi cài đặt heap ta lại dùng mảng. Các phần tử trong cây nhị phân heap được đưa vào mảng heap theo nguyên tắc "từ trên xuống dưới, từ trái sang phải". Tính chất thứ hai cũng cho thấy mảng heap được đổ đầy liên tục, không có ô trống.

Tác vụ trên heap thường là hoán chuyển giữa node cha và node con liên tiếp nhau trên dây chuyền cha-con. Quan hệ cha-con này thể hiện trong mảng như sau: node cha heap[k] có con trái là heap[2*k + 1] và con phải là heap[2*k+2]. Hoán chuyển dây chuyền cha-con trên một nhánh cây nhị phân heap, được thực hiện dễ dàng trên mảng giữa chuỗi các ô có chỉ số tương ứng với quan hệ trên.

1. Thao tác trên heap

a) Thêm phần tử vào heap

Phần tử được đưa vào cuối heap như node lá cuối, rồi cấu trúc lại heap. Cấu trúc lại heap là thực hiện việc chuyển node lá cuối theo dây chuyền quan hệ cha-con lên dần sao cho thuộc tính của heap được thỏa mãn, hoặc nó trở thành node gốc của cây.

heapEnter(el)

```
đặt el vào cuối heap;
while (node chứa el không phải là root) and (el > trị chứa trong node cha)
    hoán chuyển el với trị chứa trong node cha của nó;
```

b) Loại bỏ phần tử trên đỉnh heap

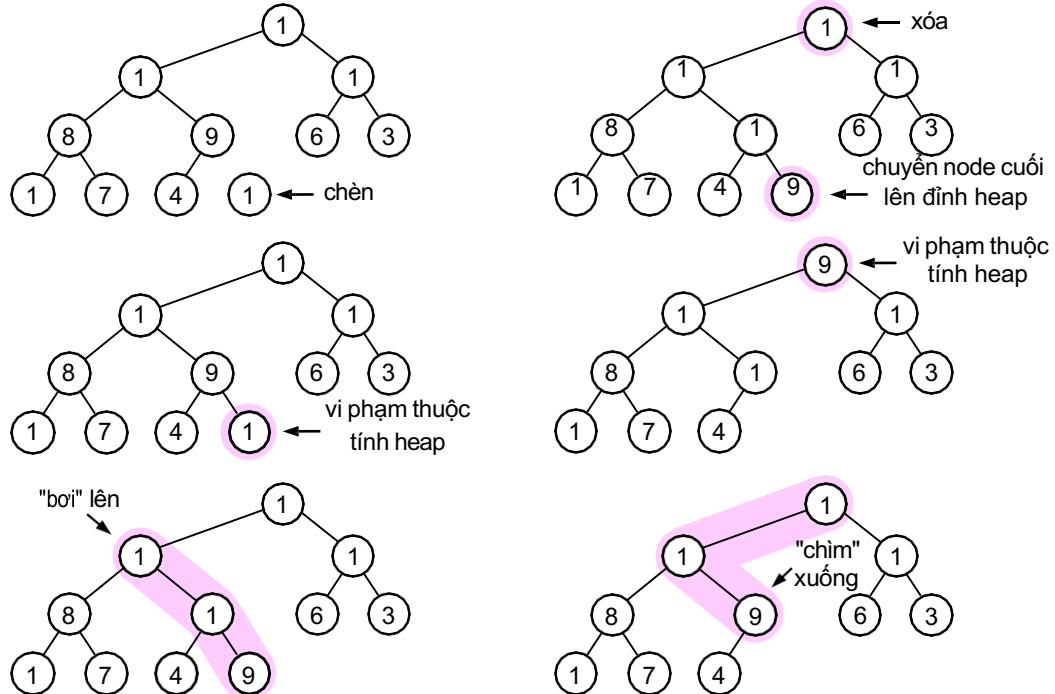
Loại bỏ phần tử đầu mảng heap, thay thế bằng phần tử cuối heap, rồi cấu trúc lại heap.

heapRemove()

```
loại bỏ phần tử tại root của heap (phần tử đầu mảng heap);
đặt trị chứa trong node lá cuối vào vị trí phần tử vừa loại bỏ;
xóa node lá cuối (phần tử cuối mảng heap);
p = root;
while (p không phải là node lá) and (trị trong p < trị chứa trong các node con của nó)
    hoán chuyển p với node con chứa trị lớn hơn;
```

Ba dòng cuối của giải thuật trên là thao tác quan trọng của heap, gọi là thao tác "chuyển xuống" moveDown:

```
void moveDown(E[] data, int first, int last) {
    int largest = 2 * first + 1;           // first là cha, largest con trái, largest+1 con phải
    while (largest <= last) {
        if (largest < last && data[largest].compareTo(data[largest + 1]) < 0)
            largest++;
        if (data[first].compareTo(data[largest]) < 0) {
            swap(data, first, largest); // hoán chuyển dữ liệu cha-con
            first = largest;          // di chuyển xuống
            largest = 2 * first + 1;
        } else largest = last + 1;     // để thoát vòng lặp, xảy ra khi tính chất heap không bị vi phạm
    }
}
```



Thêm phần tử vào heap (trái) và loại bỏ phần tử khỏi heap (phải).

Heap là một cài đặt hiệu quả cho priority queue. Khi đó:

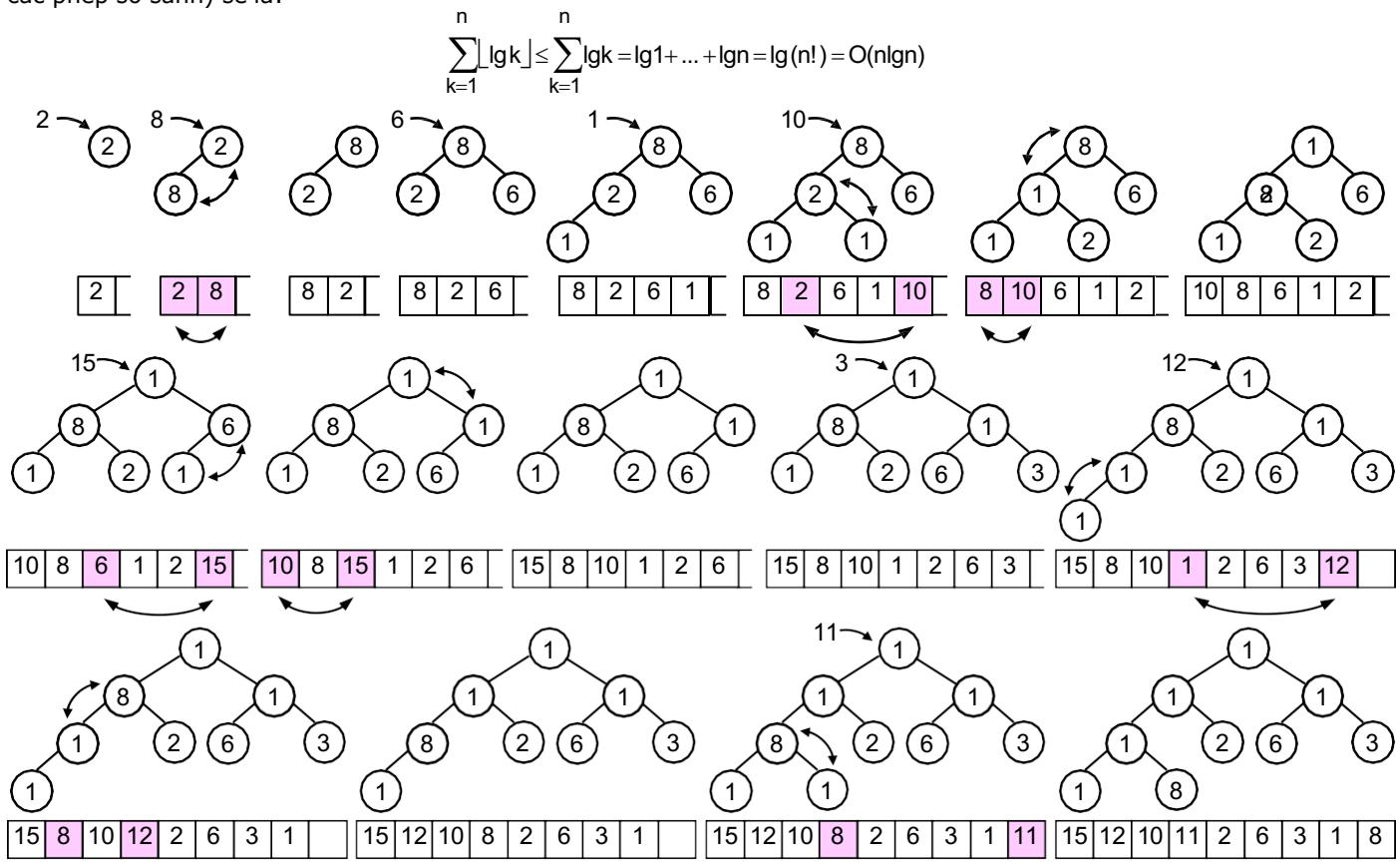
- thao tác đưa trị vào heap tương đương với enqueue, độ phức tạp $O(\lg n)$.
- lấy trị ra khỏi heap tương đương với dequeue, độ phức tạp $O(1)$.

2. Tổ chức mảng thành heap

a) Phương pháp top-down (John Williams)

Bắt đầu từ một heap rỗng, lần lượt thêm các phần tử vào heap. Nếu *vi phạm* định nghĩa heap, hoán chuyển giữa node con và node cha diễn ra trên dây chuyền cha-con ("bơi" lên).

Khi thêm một phần tử mới vào heap, trường hợp xấu nhất là phần tử này phải di chuyển lên đỉnh heap; với heap có k node, số hoán chuyển được thực hiện là $\lfloor \lg k \rfloor$. Khi thêm n phần tử vào heap, trường hợp xấu nhất, số hoán chuyển (và một số tương tự các phép so sánh) sẽ là:



Tổ chức mảng thành heap, phương pháp top-down

b) Phương pháp bottom-up (Robert Floyd)

Các heap nhỏ được tạo thành và được trộn lại với nhau tạo thành heap lớn.

FloydAlgorithm(data[]])

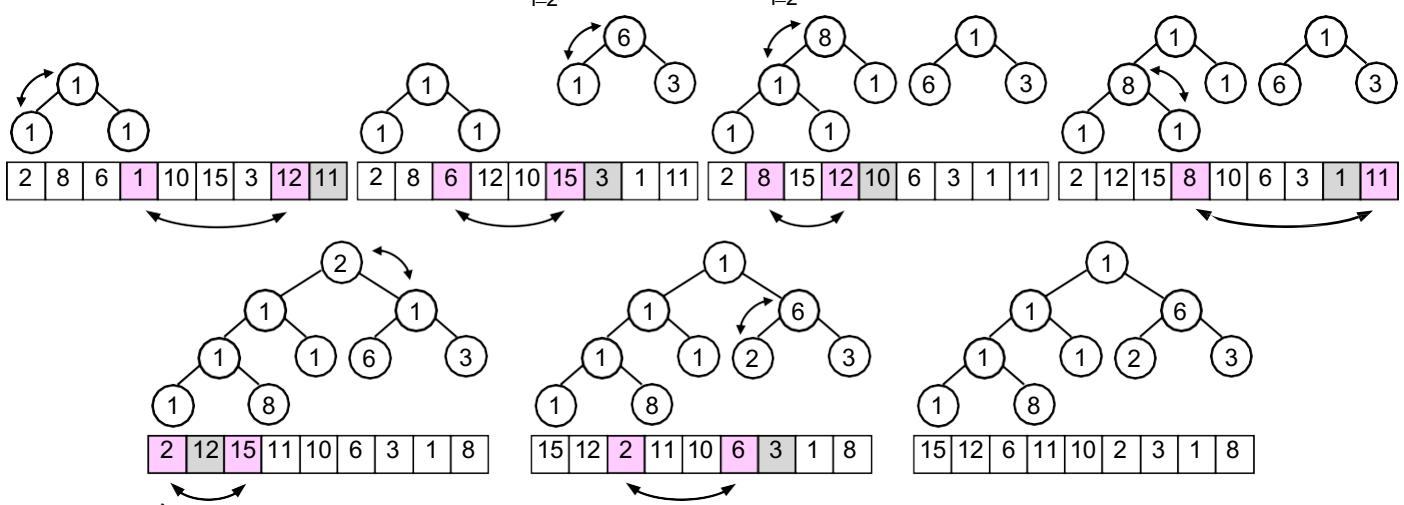
```
for (i = chỉ số của node cuối cùng không phải là lá; i > 0; i--)
```

 khôi phục tính chất heap của cây với gốc là data[i] bằng cách gọi moveDown(data, i, n-1);

Ta bắt đầu từ node cuối cùng không phải node lá, duyệt đến đầu mảng (nghĩa là ta chỉ duyệt các node trong). Node cuối cùng không phải node lá nằm tại $\text{data}[n/2 - 1]$ với n là kích thước của mảng. Nếu node đang xét nhỏ hơn một trong những node con của nó, ta hoán chuyển node đó với node con một cách dây chuyền, dồn xuống dưới cho đến khi thứ tự được thiết lập.

Giả sử rằng heap được tạo là một cây nhị phân đầy đủ, nó chứa $n = 2^k - 1$ node. Để tạo heap, moveDown() được gọi $(n + 1)/2$ lần, mỗi lần cho một node không phải node lá. Tổng số lần di chuyển là:

$$\sum_{i=2}^{\lg(n+1)} \frac{(i-1)}{2^i} = (n+1) \sum_{i=2}^{\lg(n+1)} \frac{1}{2^i}$$



Tổ chức mảng thành heap, phương pháp bottom-up. Chú ý quan hệ node cha với hai node con của nó.

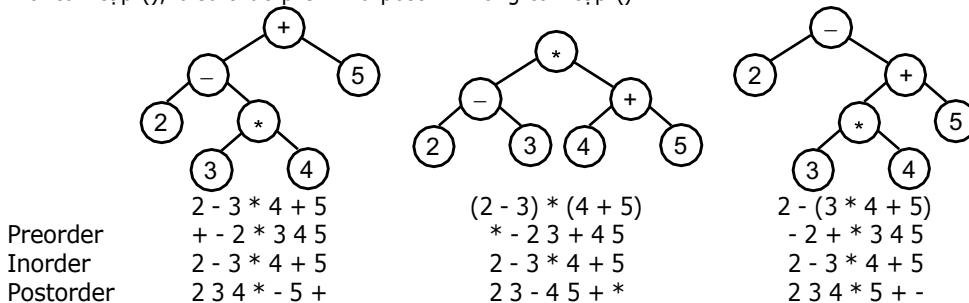
V. Cây biểu thức

Cây biểu thức là cây nhị phân có các node lá chứa các toán hạng và các node trong chứa các toán tử.

Duyệt cây biểu thức sẽ cho các dạng thể hiện khác nhau của một biểu thức:

- Duyệt theo preorder, kết quả là biểu thức prefix (tiền tố). Biểu thức prefix còn gọi là ký pháp Ba lan (Polish notation), do Jan Łukasiewicz tìm ra, cho phép tránh các cặp ngoặc () trong biểu thức infix.
- Duyệt theo inorder, kết quả là biểu thức infix (trung tố). Chú ý các cặp () phải được đưa vào để tránh nhầm lẫn.
- Duyệt theo postorder, kết quả là biểu thức postfix (hậu tố). Biểu thức postfix còn gọi là ký pháp Ba lan ngược (RPN - Reverse Polish Notation), đề xuất bởi Arthur W. Burks, Don W. Warren và Jesse B. Wright.

Chỉ biểu thức infix mới cần cặp (), biểu thức prefix và postfix không cần cặp () .



1. Xây dựng cây biểu thức từ biểu thức infix

Giải thuật tương tự phương pháp của Dijkstra (xem chương 3), dùng hai stack: stack toán tử và stack toán hạng.

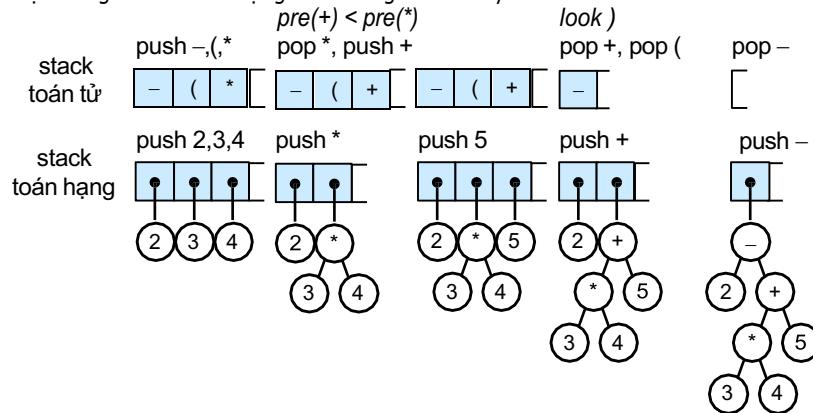
Thao tác [*]

Với mỗi toán tử lấy khỏi stack toán tử, lấy từ stack toán hạng ra hai node (theo thứ tự là y và x), sau đó tạo node mới N chứa toán tử, đặt liên kết trái của N trỏ đến x, đặt liên kết phải của N trỏ đến y. Đưa node N vào stack toán hạng.

Cây biểu thức được tạo theo phương pháp bottom-up như sau:

- Duyệt biểu thức infix từ trái sang phải.
- Nếu gặp toán hạng, tạo node chứa phần tử vừa đọc được, đưa vào stack toán hạng.
- Nếu gặp dấu mở ngoặc (, đưa vào stack toán tử.
- Nếu gặp toán tử op1:
 - + Chứng nào độ ưu tiên của toán tử op1 còn nhỏ hơn hoặc bằng độ ưu tiên của toán tử op2 trên đỉnh stack toán tử, lấy op2 ra khỏi stack toán tử. Với mỗi toán tử lấy khỏi stack toán tử, thực hiện [*].
 - + Đưa op1 vào stack toán tử.
- Nếu gặp dấu đóng ngoặc), lấy tất cả các toán tử trong stack toán tử ra cho đến khi lấy được dấu mở ngoặc (ra khỏi stack toán tử. Với mỗi toán tử lấy ra, thực hiện [*].
- Khi duyệt hết infix, lấy tất cả các toán tử nếu còn từ stack toán tử ra, thực hiện [*] với mỗi toán tử.

| - Phần tử duy nhất còn lại trong stack toán hạng chính là gốc của cây biểu thức.



Xây dựng cây biểu thức từ biểu thức infix: $2 - (3 * 4 + 5)$

Chương trình minh họa sau chuyển một biểu thức infix thành cây biểu thức (phương thức infix2Tree). Sau đó, duyệt postorder cây biểu thức này để xuất ra biểu thức postfix (phương thức tree2Postfix):

```

import java.util.*;
class TreeNode {
    String data;
    TreeNode left = null, right = null;
    public TreeNode(String data) {
        this.data = data;
    }
}

public class ExpTree {
    TreeNode root;
    public ExpTree(String infix) {
        root = ExpTree.this.infix2Tree(infix);
    }

    private void grow(String op, Stack<TreeNode> stack2) {
        TreeNode newNode = new TreeNode(op);
        newNode.right = stack2.pop();
        newNode.left = stack2.pop();
        stack2.push(newNode);
    }

    private int priority(String s) {
        if ("+-".contains(s)) return 0;
        if ("*/".contains(s)) return 1;
        return -1;
    }

    private TreeNode infix2Tree(String infix) {
        Stack<String> stack1 = new Stack<>();
        Stack<TreeNode> stack2 = new Stack<>();
        Scanner sc = new Scanner(infix);
        sc.useDelimiter(" ");
        while (sc.hasNext()) {
            if (sc.hasNextInt()) stack2.push(new TreeNode(String.valueOf(sc.nextInt()))));
            else {
                String s = sc.next();
                if ("+*/*".contains(s)) {
                    while (!stack1.empty() && priority(s) <= priority(stack1.peek())) grow(stack1.pop(), stack2);
                    stack1.push(s);
                } else if (s.equals("(")) stack1.push(s);
                else if (s.equals(")"))
                    for (String s1 = stack1.pop(); !s1.equals("(") ; s1 = stack1.pop()) grow(s1, stack2);
            }
        }
        while (!stack1.empty()) grow(stack1.pop(), stack2);
        return stack2.pop();
    }

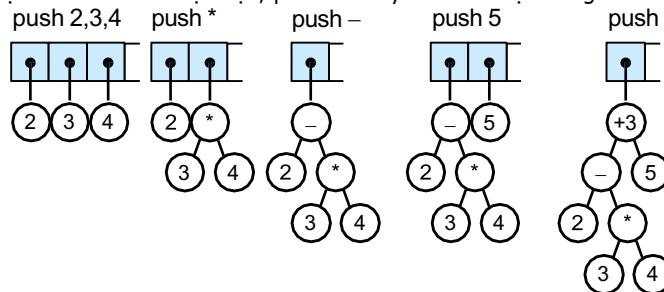
    private String LRV(TreeNode t) {
        if (t == null) return "";
        return LRV(t.left) + LRV(t.right) + t.data + " ";
    }
}
  
```

```
public String tree2Postfix() {  
    return LRV(root);  
}  
  
public static void main(String[] args) {  
    String infix = "2 - ( 3 * 4 + 5 )";  
    System.out.println("infix : " + infix);  
    System.out.println("postfix: " + new ExpTree(infix).tree2Postfix());  
}
```

2. Xây dựng cây biểu thức từ biểu thức postfix

Từ biểu thức postfix có thể khôi phục lại cây biểu thức bằng một cách đơn giản hơn. Giải thuật tạo cây biểu thức theo phương pháp bottom-up như sau:

- Khởi tạo stack rỗng.
 - Đọc lần lượt các phần tử của biểu thức postfix từ trái qua phải, phần tử này có thể là toán hạng hoặc toán tử. Với mỗi phần tử:
 - Tạo ra một node mới N chứa phần tử vừa đọc được.
 - Nếu phần tử này là một toán tử, lấy từ stack ra hai node (theo thứ tự là y và x), sau đó đặt liên kết trái của N trỏ đến x, đặt liên kết phải của N trỏ đến y.
 - Đưa node N vào stack.
 - Kết thúc bước trên, toàn bộ biểu thức đã được đọc, phần tử duy nhất còn lại trong stack chính là gốc của cây biểu thức.

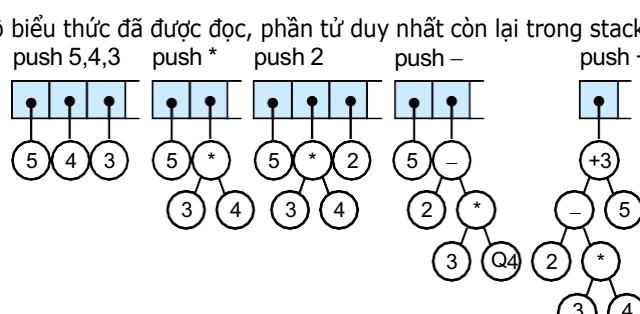


Xây dựng cây biểu thức từ biểu thức postfix: 2 3 4 * – 5 +

3. Xây dựng cây biểu thức từ biểu thức prefix

Xây dựng cây biểu thức từ biểu thức prefix cũng đơn giản, tương tự cách xây dựng cây biểu thức từ biểu thức postfix. Giải thuật như sau:

- Đảo ngược biểu thức prefix.
 - Khởi tạo stack rỗng.
 - Đọc lần lượt các phần tử của biểu thức prefix (đã đảo ngược) từ trái qua phải, phần tử này có thể là toán hạng hoặc toán tử. Với mỗi phần tử:
 - Tạo ra một node mới N chứa phần tử vừa đọc được.
 - Nếu phần tử này là một toán tử, lấy từ stack ra hai node (theo thứ tự là x và y, chú ý khác với trường hợp postfix), sau đó đặt liên kết trái của N trỏ đến x, đặt liên kết phải của N trỏ đến y.
 - Đưa node N vào stack.



Xây dựng cây biểu thức từ biểu thức prefix: + - 2 * 3 4 5
Đảo ngược trước khi xử lý, thành 5 4 3 * 2 - +

Đồ thị

I. Đồ thị

1. Khái niệm

Một **đơn đồ thị** (simple graph) $G(V, E)$ là một tập hợp *không rỗng* các đỉnh V và một tập hợp có thể rỗng các cạnh E , mỗi cạnh là tập hai đỉnh từ $V \{v_i, v_j\}$. Số các đỉnh là $|V|$ và số các cạnh là $|E|$. Nếu số cạnh nhỏ hơn $V \log V$, đồ thị gọi là *thưa* (sparse).

Một **đồ thị có hướng** (directed graph, digraph) $G(V, E)$ là một tập hợp *không rỗng* các đỉnh V và một tập hợp E các cạnh (còn gọi là cung, arc), mỗi cạnh là tập hai đỉnh từ $V \{v_i, v_j\}$ thỏa $(v_i, v_j) \neq (v_j, v_i)$.

Một **đa đồ thị** (multigraph) là đồ thị trong đó hai đỉnh có thể được nối với nhau bởi nhiều cạnh. Đa đồ thị $G(V, E, f)$ bao gồm tập đỉnh V , tập cạnh E và hàm $f: \{\{v_i, v_j\}: v_i, v_j \in V \text{ và } v_i \neq v_j\}$.

Một **đồ thị giả** (pseudograph) là một đa đồ thị không có điều kiện: $v_i \neq v_j$, nghĩa là cho phép các khuyên (loop), khuyên là cạnh nối một đỉnh với chính nó.

Một **đường đi** (path) là chuỗi các cạnh: $\text{edge}(v_1, v_2)$, $\text{edge}(v_2, v_3)$, ..., $\text{edge}(v_{n-1}, v_n)$; được ký hiệu là đường $v_1, v_2, v_3, \dots, v_{n-1}, v_n$.

Một **mạch** (circuit) là một đường đi khép kín ($v_1 = v_n$) không có cạnh nào lặp lại.

Một **chu trình** (cycle) là *một mạch* không có đỉnh nào lặp lại, ngoại trừ đỉnh đầu và đỉnh cuối.

Một **đồ thị có trọng số** (weighted graph) nếu các cạnh của nó được gán số, tùy theo tình huống sử dụng mà trọng số thể hiện chi phí, khoảng cách, chiều dài, ...

Một **đồ thị đầy đủ** (complete) nếu giữa hai đỉnh bất kỳ luôn luôn có cạnh nối.

Hai đỉnh v_i và v_j được gọi là *kề* (adjacent) nếu cạnh $v_i v_j$ có trong tập E , cạnh đó gọi là cạnh *liên thuộc* (incident) với v_i và v_j .

Bậc của đỉnh v , ký hiệu $\deg(v)$ là số cạnh liên thuộc với v . Nếu $\deg(v) = 0$ thì v được gọi là *đỉnh cô lập* (isolated vertex).

2. Biểu diễn đồ thị

a) Biểu diễn bằng danh sách kề

Danh sách kề (adjacency list) xác định tất cả các đỉnh kề với mỗi đỉnh trong đồ thị. Cài đặt bằng danh sách liên kết hoặc mảng.

b) Biểu diễn bằng ma trận

- Ma trận kề (adjacency matrix) của một đồ thị $G(V, E)$ là một ma trận $|V| \times |V|$ sao cho mỗi phần tử của ma trận:

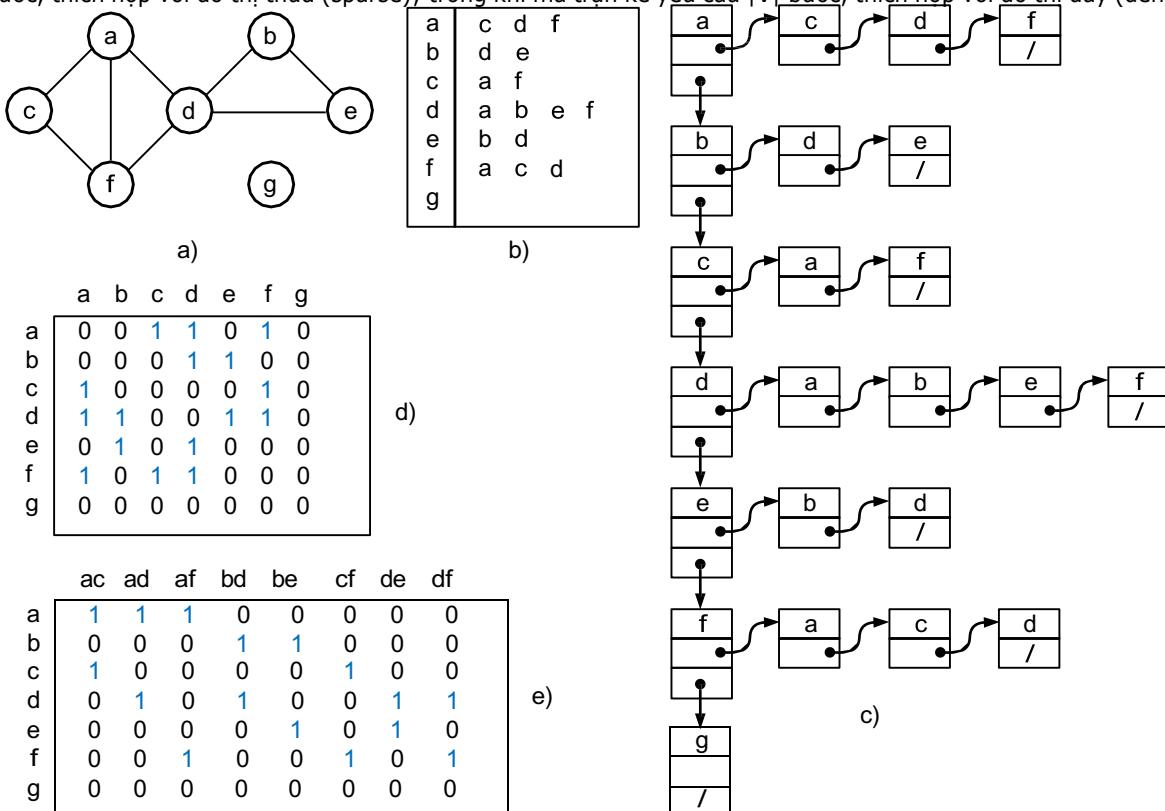
$$a_{ij} = \begin{cases} 1 & \text{nếu tồn tại cạnh } \text{edge}(v_i v_j) \\ 0 & \text{ngược lại} \end{cases}$$

Ma trận kề có thể mở rộng với đa đồ thị, a_{ij} = số cạnh giữa v_i và v_j ; hoặc với đồ thị có trọng số, a_{ij} = trọng số cạnh $v_i v_j$.

- Ma trận liên thuộc (incidence matrix) của một đồ thị $G(V, E)$ là một ma trận $|V| \times |E|$ sao cho mỗi phần tử của ma trận:

$$a_{ij} = \begin{cases} 1 & \text{nếu cạnh } e_j \text{ có liên thuộc với đỉnh } v_i \\ 0 & \text{ngược lại} \end{cases}$$

Cách biểu diễn đồ thị tùy theo yêu cầu của vấn đề cần giải quyết. Ví dụ yêu cầu là xử lý các đỉnh kề của v , danh sách kề yêu cầu bước, thích hợp với đồ thị thưa (sparse); trong khi ma trận kề yêu cầu $|V|$ bước, thích hợp với đồ thị dày (dense).



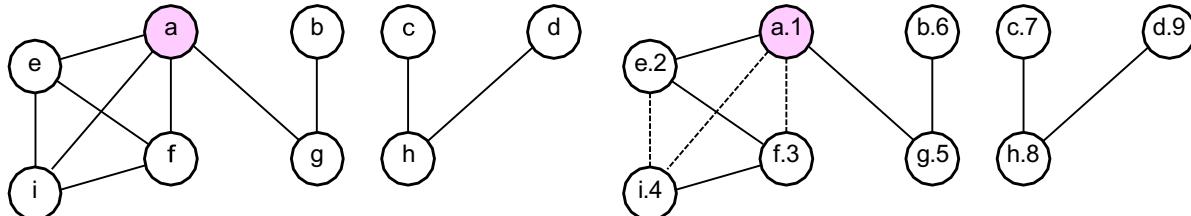
Biểu diễn đồ thị a) bằng b-c) danh sách kề, d) ma trận kề, e) ma trận liên thuộc

II. Duyệt đồ thị

Duyệt đồ thị là thăm các đỉnh của đồ thị, mỗi đỉnh một lần. Không thể dùng giải thuật duyệt cây để duyệt đồ thị vì trong đồ thị có chu trình, áp dụng giải thuật duyệt cây có thể tạo ra vòng lặp vô hạn. Để tránh điều đó, mỗi đỉnh được thăm được đánh dấu để tránh thăm lại đỉnh đó. Ngoài ra, đồ thị cũng có thể có nhiều thành phần liên thông, cần bảo đảm đã thăm tất cả các đỉnh để tránh bỏ sót các thành phần liên thông khác.

1. Duyệt theo chiều sâu (DFS - Depth-First Search) (John Hopcroft và Robert Tarjan)

Mỗi đỉnh v được thăm và sau đó mỗi đỉnh kề với v mà chưa được thăm sẽ được thăm kế tiếp. Nếu đỉnh v không có đỉnh kề hoặc tất cả các đỉnh kề của nó đều đã được thăm, ta quay lui lại đỉnh được thăm trước v . Duyệt kết thúc nếu việc duyệt và quay lui này dẫn đến đỉnh đầu tiên khi bắt đầu duyệt. Nếu vẫn còn đỉnh chưa thăm trong đồ thị, duyệt bắt đầu lại với một trong những đỉnh chưa thăm.



Thứ tự duyệt DFS, các đường liền là tập edges (cạnh tiến)

DFS(v)

```

num(v) = i++;
for tất cả những đỉnh u kề với v
    if (num(u) == 0)
        thêm edge(uv) vào edges;
        DFS(u);
    
```

depthFirstSearch()

```

for tất cả những đỉnh v
    num(v) = 0;
    edges = null;
    i = 1;
    while có một đỉnh v mà num(v) == 0 // duyệt các thành phần liên thông
        DFS(v);
        hiển thị edges;
    
```

Chú ý:

- $\text{num}(v) > 0$ cho biết đỉnh đã thăm, hơn nữa, nó cho biết thứ tự thăm. $\text{num}(v) < \text{num}(u)$ nghĩa là v được thăm trước u .
- Vòng lặp for trong $\text{DFS}(v)$ cho phép quay lui về v để thăm tiếp một đỉnh kề khác của v .
- DSF() chỉ duyệt một thành phần liên thông, vòng lặp while trong $\text{depthFirstSearch}()$ cho phép duyệt tất cả các thành phần liên thông của đồ thị, nếu có.

Giải thuật duyệt theo chiều sâu sinh ra một cây khung hoặc một tập cây khung, gọi là rừng (forest). Cây khung của một thành phần liên thông là cây trải ra trên tất cả các đỉnh của thành phần liên thông đó. Các cạnh của cây khung gọi là **cạnh tiến** (forward edges, nối từ đỉnh duyệt trước đến đỉnh duyệt sau) hoặc **cạnh DFS**, các cạnh còn lại (đường đứt nét) gọi là **cạnh lùi** (backward edges, nối từ đỉnh duyệt sau đến đỉnh duyệt trước). Cạnh lùi quan trọng trong các giải thuật về tính liên thông của đồ thị.

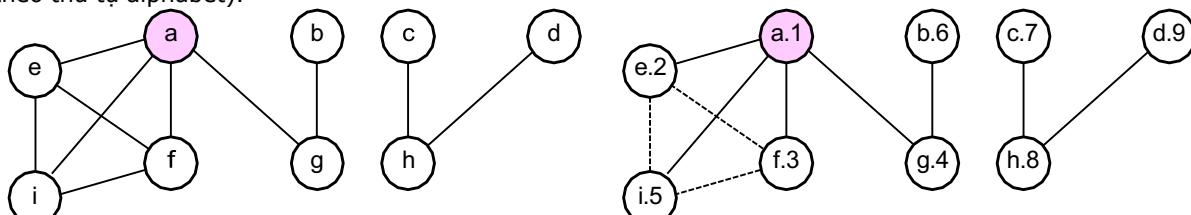
Độ phức tạp của duyệt theo chiều sâu là $O(|V|+|E|)$:

- Khởi tạo $\text{num}(v)$ bằng 0 cho mỗi đỉnh yêu cầu $|V|$ bước.
- $\text{DFS}(v)$ được gọi $\text{deg}(v)$ lần cho mỗi đỉnh v , nghĩa là một lần cho một cạnh của v , vì thế tổng số lời gọi là $2|E|$.

2. Duyệt theo chiều rộng (BFS - Breadth-First Search)

Có thể khử đê quy trong DFS bằng cách dùng stack. Nếu dùng queue, ta có thể duyệt theo chiều rộng đồ thị giống như duyệt cây, chỉ chú ý đánh dấu các đỉnh để tránh duyệt lại đỉnh đã duyệt.

Khi thăm một đỉnh, tất cả các đỉnh kề với đỉnh đó và chưa thăm sẽ được đưa vào queue theo một thứ tự quy định trước (trong ví dụ, theo thứ tự alphabet).



Thứ tự duyệt BFS, các đường liền là tập edges

breadthFirstSearch()

```

for tất cả những đỉnh v
    num(v) = 0;
    edges = null;
    i = 1;
    while có một đỉnh v mà num(v) == 0
        num(v) = i++;
        
```

```

enqueue(v);
while queue không rỗng
    v = dequeue();
    for tất cả đỉnh u kề với v
        if num(u) == 0
            num(u) = i++;
            enqueue(u);
            thêm edge(vu) vào edges;
    hiển thị edges;

```

Chú ý:

- Vòng lặp while bên trong (xử lý queue) duyệt theo chiều rộng một thành phần liên thông. Queue này được khởi tạo bằng đỉnh đầu tiên của thành phần liên thông đó, đỉnh đầu tiên được đánh dấu và đưa vào queue trước khi duyệt.

- Khi thăm một đỉnh, đỉnh đó được đưa ra khỏi queue, tất cả các đỉnh kề với nó và chưa thăm được đưa vào queue.

- Vòng lặp while bên ngoài cho phép duyệt tất cả các thành phần liên thông của đồ thị, nếu có.

Độ phức tạp của BFS là $O(|V| + |E|)$ nếu đồ thị được biểu diễn bằng danh sách kề, bằng $O(|V|^2)$ nếu đồ thị được biểu diễn bằng ma trận kề.

III. Đường đi ngắn nhất

1. Giải thuật tổng quát

Giải thuật tìm đường đi ngắn nhất được thực hiện trên đồ thị có trọng số. Trọng số thể hiện khoảng cách giữa hai đỉnh, thời gian di chuyển, chi phí truyền thông, ...

Khi xác định đường đi ngắn nhất giữa hai đỉnh u và v, thông tin về khoảng cách giữa các điểm trung gian w phải được ghi lại, như một nhãn (label) gắn liền với các đỉnh này. Tùy theo việc cập nhật các nhãn, các phương pháp tìm đường đi ngắn nhất chia làm hai lớp giải thuật:

- Phương pháp thiết lập nhãn (label setting): mỗi lần đi qua đỉnh, nhãn của đỉnh được thiết lập *một trị không đổi* cho đến cuối quá trình thực thi. Hạn chế là chỉ xử lý các đồ thị có trọng số dương.

- Phương pháp chỉnh nhãn (label correcting): cho phép thay đổi nhãn của đỉnh trong quá trình thực thi. Xử lý được đồ thị có trọng số âm nhưng không được có chu trình âm (tổng trọng số các cạnh trong chu trình là số âm).

Gallo và Pallottino đề xuất *giải thuật tổng quát* cho phép tìm đường đi ngắn nhất từ một đỉnh đến các đỉnh còn lại:

genericShortestPathAlgorithm(đồ thị có trọng số, đỉnh first)

```

for tất cả đỉnh v
    currDist(v) = ∞;
    currDist(first) = 0;
khởi tạo toBeChecked; (1)
while toBeChecked không rỗng
    v = một đỉnh trong toBeChecked; (2)
    loại bỏ v khỏi toBeChecked;
    for tất cả đỉnh u kề với v
        if currDist(v) + weight(edge(vu)) < currDist(u)
            currDist(u) = currDist(v) + weight(edge(vu));
            predecessor(u) = v;
            thêm u vào toBeChecked nếu nó không có trong đó;

```

Giải thuật tổng quát có hai vấn đề mở: tổ chức của cấu trúc dữ liệu toBeChecked (1) và cách chọn đỉnh v từ toBeChecked (2). Các lớp giải thuật trình bày tiếp theo sau phân biệt chủ yếu ở hai điểm này.

2. Giải thuật Dijkstra (single-source shortest-paths)

Trong giải thuật Dijkstra, một số đường đi p_1, \dots, p_n từ đỉnh v được kiểm tra để chọn ra đường đi nhỏ nhất giữa chúng, nghĩa là mỗi đường đi p_i có cơ hội được thêm một cạnh. Nhưng nếu p_i dài hơn một đường đi được kiểm tra khác, p_i sẽ bị bỏ qua và đường đi khác đó được tiếp tục bằng cách thêm một cạnh vào đó.

Vì đường đi có thể dẫn đến đỉnh có nhiều cạnh đi ra, nên những đường đi mới (tương ứng với các cạnh đi ra) sẽ làm tăng thêm số các đường đi phải xét. Mỗi đỉnh chỉ được kiểm tra một lần, tất cả các đường đi ra từ nó đều mở và sau đó đỉnh sẽ bị loại bỏ khỏi toBeChecked và không sử dụng nữa.

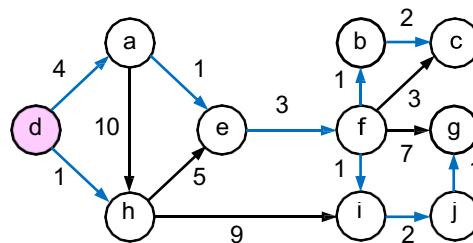
dijkstraAlgorithm(đồ thị có trọng số, đỉnh first)

```

for tất cả đỉnh v
    currDist(v) = ∞; // nhãn của v
    currDist(first) = 0;
khởi tạo toBeChecked = tất cả các đỉnh;
while toBeChecked không rỗng
    v = một đỉnh trong toBeChecked với currDist(v) nhỏ nhất; // cụ thể cách chọn đỉnh trong toBeChecked
    loại bỏ v khỏi toBeChecked;
    for tất cả đỉnh u kề với v và nằm trong toBeChecked
        if currDist(v) + weight(edge(vu)) < currDist(u)
            currDist(u) = currDist(v) + weight(edge(vu));
            predecessor(u) = v;

```

Chú ý nhãn của các đỉnh bị loại khỏi toBeChecked được thiết lập một cách vĩnh viễn, cho thấy đây là giải thuật thiết lập nhãn.



Có 10 vòng lặp để xét 10 đỉnh trong toBeChecked . Các đỉnh được khởi tạo với nhãn rất lớn ∞ , ngoại trừ d là điểm khởi đầu nên khởi tạo với trị 0. d được chọn và loại khỏi toBeChecked , nhãn của các đỉnh kề d (a, h đều có nhãn là ∞) được gán bằng nhãn của $d +$ trọng số các cạnh từ d . Trong toBeChecked , h có nhãn nhỏ nhất nên được chọn; nó bị loại khỏi toBeChecked , đồng thời gán nhãn hai đỉnh kề h là e và i , a không gán nhãn lại do nhãn của $h +$ trọng số cạnh ha lớn hơn nhãn của a . Bây giờ, đỉnh có nhãn nhỏ nhất trong toBeChecked là a , nên a được chọn và loại khỏi toBeChecked , cập nhật nhãn cho e, \dots . Cứ như thế, cho đến khi toBeChecked rỗng và giải thuật kết thúc.

Giải thuật Dijkstra trả về một cây khung, được tạo từ các đỉnh đã chọn và predecessor của chúng, predecessor của đỉnh v là đỉnh thiết lập nhãn cho v . Ví dụ: nhãn của đỉnh e thiết lập lần cuối là 5 với $\text{predecessor}(e) = a$, ae có trong cây khung.

Định đang xét:	Lần lặp: khởi tạo	1 2 3 4 5 6 7 8 9 10									
		d	h	a	e	f	b	i	c	j	g
a	∞		4	4							
b	∞		∞	∞	∞	∞	9				
c	∞		∞	∞	∞	∞	11	11	11		
d	0										
e	∞		∞	6	5						
f	∞		∞	∞	∞	8					
g	∞		∞	∞	∞	15	15	15	15	12	
h	∞	1									
i	∞		10	10	10	9	9				
j	∞		∞	∞	∞	∞	∞	11	11		

Độ phức tạp của giải thuật Dijkstra là $O(|V|^2)$:

- Vòng lặp for và while đầu tiên thực hiện $|V|$ lần.
- Trong mỗi vòng của vòng lặp while:
 - + tìm đỉnh v có $\text{currDist}(v)$ nhỏ nhất trong toBeChecked yêu cầu $O(|V|)$
 - + vòng lặp for duyệt các đỉnh kề thực hiện $\text{deg}(v)$ lần, cũng yêu cầu $O(|V|)$.

Có thể sử dụng min heap, một loại hàng đợi ưu tiên, làm toBeChecked để hiệu quả của giải thuật tăng lên, độ phức tạp là $O((|E|+|V|)\lg|V|)$.
Giải thuật không đúng trong trường hợp trọng số âm được dùng trong đồ thị.

3. Giải thuật Ford

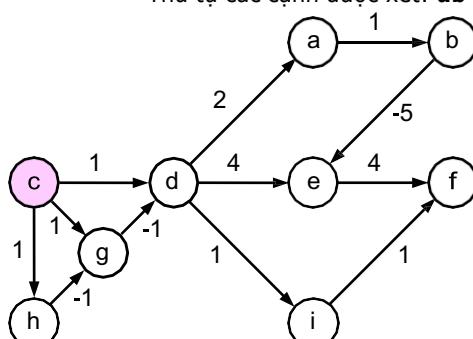
Giải thuật Ford cũng thiết lập nhãn với khoảng cách hiện thời, nhưng không xác lập vĩnh viễn cho bất kỳ đỉnh nào, mà vẫn hiệu chỉnh nhãn cho đến khi xử lý xong toàn bộ đồ thị, cho thấy đây là giải thuật chỉnh nhãn. Giải thuật Ford có thể làm việc với đồ thị có trọng số âm, nhưng đồ thị không được có chu trình âm.

fordAlgorithm(**đồ thị có trọng số, đỉnh first**)

```

for tất cả đỉnh v
  currDist(v) =  $\infty$ ;
  currDist(first) = 0;
while có cạnh edge(vu) sao cho currDist(v) + weight(edge(vu)) < currDist(u)
  currDist(u) = currDist(v) + weight(edge(vu));
  
```

Thứ tự các cạnh được xét: **ab be cd cg ch da de di ef gd hg if**



khởi tạo	Lần lặp			
	1	2	3	4
a	∞	3	3	2
b	∞	∞	∞	4
c	0			3
d	∞	1	0	-1
e	∞	5	5	-1
f	∞	9	3	2
g	∞	1	0	1
h	∞	1		
i	∞	2	2	1

Giải thuật duyệt qua tất cả các cạnh để tìm cách cải thiện nhãn (currDist) của đỉnh tới, chuỗi các cạnh phải duyệt sắp theo thứ tự alphabet. Mỗi một vòng lặp là một lần duyệt qua các cạnh, nhãn đỉnh tới của cạnh có thể thay đổi vài lần trong vòng lặp. Khi không còn cập nhật đỉnh, giải thuật kết thúc; mỗi đỉnh chứa nhãn cho thấy đường đi ngắn nhất từ đỉnh bắt đầu đến nó.

Ví dụ trong hình trên, lần lặp 1, đỉnh d có nhãn 1 khi duyệt cạnh cd nên a có nhãn 3 khi duyệt cạnh da sau đó; tuy nhiên do g có nhãn 1 (duyệt cạnh cg), nên khi duyệt đến cạnh gd đỉnh d được chỉnh nhãn thành 0.

Độ phức tạp của giải thuật là $O(|V||E|)$: Có nhiều nhất $|V|-1$ lần duyệt qua chuỗi có $|E|$ cạnh. Trong lần duyệt thứ nhất, tất cả đường đi một cạnh được xác định, trong lần duyệt thứ hai, tất cả đường đi hai cạnh được xác định, và cứ tiếp tục như vậy. Với đồ thị có trọng số vô tỷ, độ phức tạp là $O(2^{|V|})$ (Gallo và Pallottino, 1986).

4. Giải thuật chỉnh nhãn tổng quát

Giống như giải thuật Dijkstra, có thể cải thiện hiệu quả giải thuật Ford bằng cách duyệt các cạnh và đỉnh theo một thứ tự nào đó. Ta cũng nhận thấy trong mỗi vòng lặp mọi cạnh phải được kiểm tra dù không cần thiết. Có thể cải tiến điều này bằng giải thuật chỉnh nhãn, dựa trên giải thuật tìm đường đi ngắn nhất tổng quát.

labelCorrectingAlgorithm(*đồ thị có trọng số, đỉnh first*)

```

for tất cả đỉnh v
    currDist(v) = ∞;
    currDist(first) = 0;
    toBeChecked = {first};
    while toBeChecked không rỗng
        v = một đỉnh trong toBeChecked;
        loại bỏ v khỏi toBeChecked;
        for tất cả đỉnh u kề với v
            if currDist(v) + weight(edge(vu)) < currDist(u)
                currDist(u) = currDist(v) + weight(edge(vu));
                predecessor(u) = v;
                thêm u vào toBeChecked nếu nó không có trong đó;
    
```

Hiệu quả của giải thuật này liên quan đến cấu trúc dữ liệu dùng cho toBeChecked.

Một cách là dùng queue: đỉnh v được lấy ra khỏi queue, bất kỳ đỉnh u nào kề nó có nhãn *được cập nhật* (và chưa có trong queue) sẽ được đưa vào queue (đỉnh tô đậm).

Định đang xét																							
queue	c	d	g	h	a	e	i	d	g	b	f	a	e	i	d	b	f	a	i	e	b	f	e
	d	g	h	a	e	i	d	g	b	f	a	e	i	d	b	f	a	i	e	b	f	e	
	g	h	a	e	i	d	g	b	f	a	e	i	d	b	f	a	i	e	b	f	e		
	h	a	e	i	d	g	b	f	a	e	i	d	b	f	a	i	e	b	f				
	e	i	d	g	b	f	a	e	i	d	b		i	e									
	i	d	g	b	f		e	i	d														
A	∞	∞	3	3	3			2	2	2	2	2						1	1	1			
B	∞	∞	∞	∞	∞	4	4	4	4	4		3	3	3	3			2	2	2			
C	0																						
D	∞	1	0	0	0	0	0			-1	-1	-1	-1	-1	-1								
E	∞	∞	5	5	5	5			4	4	1	-1	-1						2	-2	-2	-2	3
F	∞	∞	∞	∞	∞	∞	9	3	3	3	3					2	2	2		1	1	1	
G	∞	1	1	0	0	0	0	0															
H	∞	1	1	1																			
I	∞	∞	2	2	2	2	2		1	1	1	1	1	1	0	0	0	0					

Ta thấy nhãn của d được cập nhật ba lần, những cập nhật này gây nên thay đổi dây chuyền đến các đỉnh sau nó. Để tránh những vòng lặp không cần thiết khi cập nhật, ta dùng queue hai đầu, còn gọi là deque: các đỉnh được thêm vào toBeChecked *lần đầu* sẽ được đặt vào cuối queue (last), ngược lại chúng sẽ được đặt vào đầu queue (first).

Điều này dựa trên nhận xét: nếu đỉnh v được xử lý lần đầu thì các đỉnh theo sau nó chưa từng được xử lý nên chúng sẽ được xử lý sau v. Ngược lại, nếu v *đã được xử lý* thì các đỉnh theo sau nó có thể vẫn còn chờ trong queue; đặt v vào đầu queue giúp cho các đỉnh theo sau nó nhận được cập nhật mới nhất từ nó.

Định đang xét																							
deque	c	d	g	d	h	g	d	a	e	i	b	e	f										
	d	g	d	h	g	d	a	e	i	b	e	f											
	g	h	h	a	a	a	e	i	b	f	f												
	h	a	a	e	e	e	i	b	f														
	e	e	i	i	i	i																	
a	∞	∞	3	3	2	2	2	2	1														
b	∞	∞	∞	∞	∞	∞	∞	∞	∞	2	2	2											
c	0																						
d	∞	1	0				-1																
e	∞	∞	5	5	4	4	4	3	3				-3										
f	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	7	1	1	1							
g	∞	1	1			0																	
h	∞	1	1	1	1																		
i	∞	∞	2	2	1	1	1	0	0	0													

5. Giải thuật WFI (Stephen Marshall, Robert W. Floyd và P.Z. Ingberman)

Giải thuật tìm đường đi ngắn nhất từ một đỉnh bất kỳ đến các đỉnh còn lại (All-to-All) được cài đặt bằng *ma trận kề-trọng số*, đồ thị có thể chứa trọng số âm. Ma trận kề-trọng số là ma trận kề, nhưng giao giữa hàng (mô tả đỉnh 1) và cột (mô tả đỉnh 2) của một cạnh không phải là 1 mà là trọng số của cạnh đó. Giải thuật như sau:

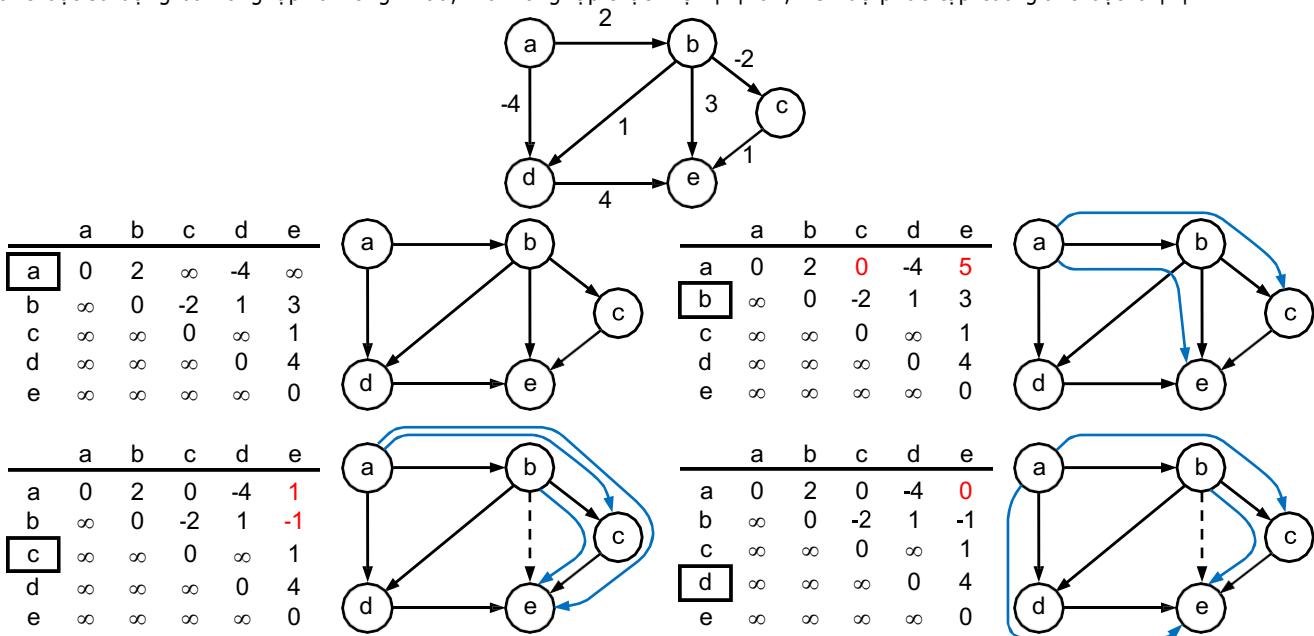
wfiAlgorithm(*ma trận kề-trọng số*)

```
for i = 1 to |V|
  for j = 1 to |V|
    for k = 1 to |V|
      if weight[j][k] > weight[j][i] + weight[i][k]
        weight[j][k] = weight[j][i] + weight[i][k];
```

Hai vòng lặp trong, theo j và theo k, duyệt tất cả các đường đi từ đỉnh j đến đỉnh k có chiều dài là $\text{weight}[j][k]$. Vòng lặp ngoài cùng theo i, tham chiếu đến các đỉnh i có thể *nằm giữa* đỉnh j và đỉnh k. Như vậy, tất cả các đường đi $v_j \dots v_i \dots v_k$ (từ từ đỉnh j đến đỉnh k, có qua đỉnh i) thì $\text{weight}[j][k]$ sẽ được cập nhật với trị nhỏ hơn. Một đường đi tốt hơn sẽ luôn được chọn nếu có thể, ví dụ đường đi trực tiếp từ b đến e sẽ bị loại bỏ khi ta tìm được đường đi tốt hơn từ b đến e thông qua c.

Giải thuật WFI cũng cho phép *phát hiện chu trình* nếu đường chéo được khởi tạo ∞ thay vì 0. Bất cứ phần tử nào trên đường chéo bị thay đổi sẽ là dấu hiệu cho thấy đồ thị có chu trình.

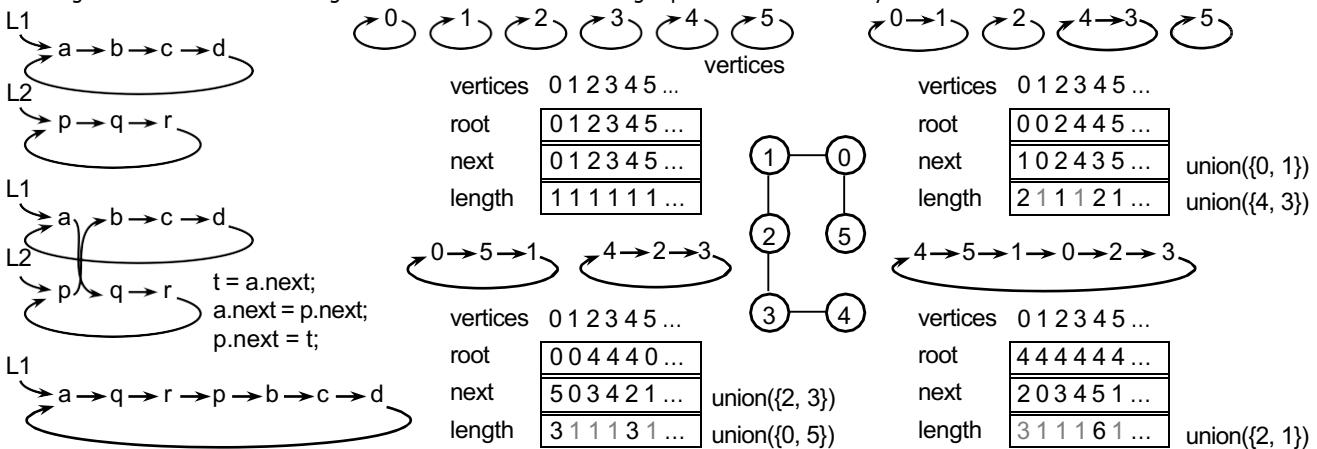
Giải thuật sử dụng ba vòng lặp for lồng nhau, mỗi vòng lặp thực hiện $|V|$ lần, nên độ phức tạp của giải thuật là $|V|^3$.



IV. Phát hiện chu trình

Bài toán tìm kiếm hợp nhất (Union-Find): lúc đầu, mỗi đỉnh trong đồ thị được lưu trong một danh sách riêng có root là chính đỉnh đó. Thực hiện union(vu) nghĩa là ta ghép danh sách chứa đỉnh v và danh sách chứa đỉnh u thành một danh sách lớn hơn. Điều kiện thực hiện union(vu): cạnh vu tồn tại và v, u thuộc hai danh sách khác nhau.

Ta dùng danh sách liên kết vòng để lưu danh sách các đỉnh ghép các danh sách này.



Trái: ghép hai tập đỉnh bằng cách kết nối hai danh sách liên kết vòng. Phải: gọi union(vu) nếu cạnh vu tồn tại

initialize()

```
for i = 0 to |V| - 1
  root[i] = next[i] = i
  length[i] = 1
```

union(edge(vu))

```
if (root[u] == root[v])
  // loại bỏ cạnh nếu u và v cùng một tập
```

```

    return;
else if (length[root[v]] < length[root[u]]) // ghép hai tập thành một
    rt = root[v];
    length[root[u]] += length[rt];
    root[rt] = root[u];
    for (j = next[rt]; j != rt; j = next[j])
        root[j] = root[u]; // cập nhật root của rt và các đỉnh khác trong danh sách
    swap(next[rt], next[root[u]]); // ghép hai danh sách
    thêm edge(vu) vào spanningTree;
else // nếu length[root[v]] ≥ length[root[u]]
    xử lý như trước, đảo ngược u và v.

```

Bài toán union-find quan trọng, vì giúp giải quyết nhiều vấn đề trong đồ thị:

- Xây dựng cây khung: khi thực hiện union() với các cạnh (thực hiện union(vu) khi xử lý cạnh vu), ta loại bỏ các cạnh với *hai đỉnh đã có trong danh sách* và lưu các cạnh được xử lý. Các cạnh được lưu sẽ thuộc về *cây khung* của đồ thị.
- Phát hiện chu trình: cạnh bị loại bỏ ở trên sẽ thuộc về một chu trình. Vì giữa hai đỉnh của cạnh đó còn hai đường đi khác nhau: một có trong danh sách và một là cạnh loại bỏ.

V. Cây khung (spanning tree)

Cây khung, còn gọi là cây bao trùm, của một đồ thị liên thông là cây trải ra trên *tất cả các đỉnh* của đồ thị. Bài toán được quan tâm là bài toán tìm cây khung nhỏ nhất, là cây khung có *tổng trọng số các cạnh* là nhỏ nhất.

Giải thuật tìm cây khung nhỏ nhất có nhiều lời giải, chia làm một số nhóm:

- Xây dựng cây khung bằng cách thêm cách nhánh mới đến nó, có thể loại bỏ nhánh nếu cần (giải thuật Dijkstra).
- Mở rộng tập các cây để tạo thành một cây khung (giải thuật Kruskal).

1. Giải thuật Dijkstra

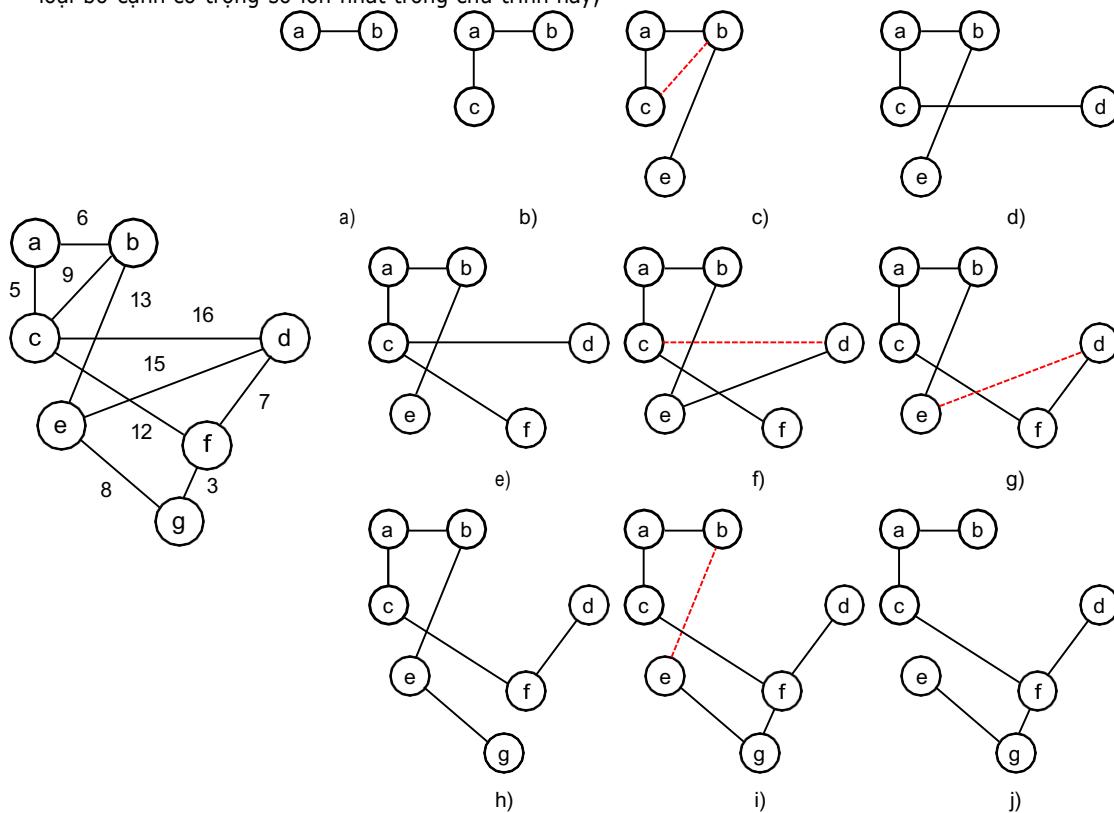
Cây khung được mở rộng bằng cách thêm từng cạnh một và nếu phát hiện có chu trình thì cạnh có trọng số lớn nhất trong chu trình đó sẽ bị loại bỏ.

dijkstraMethod(*đồ thị vô hướng* liên thông có trọng số)

```

tree = null;
edges = chuỗi các cạnh của đồ thị, không cần sắp xếp;
for i = 1 to |edges|
    thêm ei vào tree;
    if có chu trình trong cây
        loại bỏ cạnh có trọng số lớn nhất trong chu trình này;

```



Giải thuật Dijkstra. Chọn cạnh theo thứ tự alphabet. Cạnh loại bỏ là cạnh có nét đứt.

Phương pháp của Dijkstra dùng một phiên bản biến đổi của union() để phát hiện chu trình. Thời gian thực hiện là $O(|E||V|)$.

2. Giải thuật Kruskal

Tất cả các cạnh được sắp xếp theo trọng số tăng dần. Sau đó mỗi cạnh, theo thứ tự sắp xếp, được thêm vào cây nếu việc thêm nó vào không tạo ra chu trình.

kruskalAlgorithm(*đồ thị vô hướng* liên thông có trọng số)

```

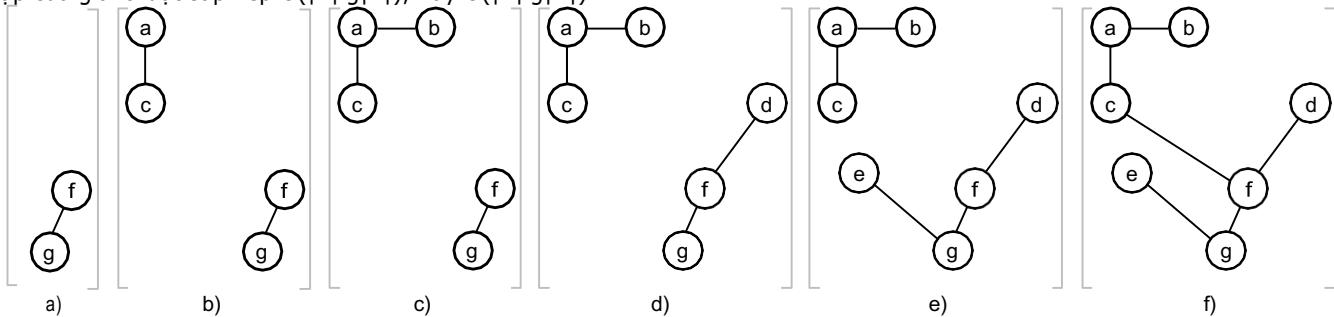
tree = null;
edges = chuỗi các cạnh của đồ thị, sắp xếp theo trọng số tăng dần;
for (i = 1; i <= |E| and |tree| < |V| - 1; i++)           // |tree| là số cạnh của cây khung đang tạo
    if ei thuộc edges và không tạo chu trình với các cạnh trong tree
        thêm ei vào tree;
    
```

Độ phức tạp của giải thuật bằng độ phức tạp của giải thuật sắp xếp được áp dụng; nếu sắp xếp hiệu quả, độ phức tạp sẽ là $O(|E|\lg|E|)$. Ngoài ra, nó cũng phụ thuộc vào phương pháp sử dụng để phát hiện chu trình. Nếu ta sử dụng union() để cài đặt thì vòng lặp for trở thành:

```

for (i = 1; i <= |E| and |tree| < |V| - 1; i++)
    union(ei = edges(vu))
    
```

union() được gọi |E| lần, nó thực hiện phép hợp với độ phức tạp $O(|V|)$, để thêm $|V| - 1$ cạnh vào cây khung. Suy ra, độ phức tạp của vòng lặp là $O(|E| + (|V| - 1)|V|)$, tức $O(|V|^2)$. Như vậy, độ phức tạp của giải thuật Kruskal được xác định bởi độ phức tạp của giải thuật sắp xếp $O(|E|\lg|E|)$, hay $O(|E|\lg|V|)$.



Giải thuật Kruskal.

Thứ tự thêm các cạnh: fg, ac, ab, df, eg, (bc), cf, (be), (de), (cd). Cạnh trong cặp () là cạnh bị loại bỏ.

VI. Tính liên thông (Connectivity)

1. Tính liên thông trong đồ thị vô hướng

Đồ thị vô hướng được gọi là liên thông nếu có đường đi giữa hai đỉnh bất kỳ thuộc đồ thị. Một đồ thị liên thông nhiều hay ít tùy thuộc vào số đường đi khác nhau (không có đỉnh chung) giữa các đỉnh. Đồ thị liên thông bậc n (n-connected) nếu có ít nhất n đường đi khác nhau giữa hai đỉnh bất kỳ.

Ta đặc biệt quan tâm đồ thị 2-connected. Nếu có một đỉnh, gọi là đỉnh x, luôn nằm trên đường đi giữa hai đỉnh bất kỳ a và b thì đồ thị không 2-connected. Nếu loại đỉnh x khỏi đồ thị, sẽ không thể tìm được đường đi từ a đến b, đồ thị bị tách ra thành hai đồ thị con. Đỉnh x như vậy gọi là đỉnh khớp nối (articulation) hoặc đỉnh cắt (cut vertex). Nếu một cạnh mà khi loại bỏ nó làm cho đồ thị tách thành hai đồ thị con thì cạnh đó gọi là cạnh cầu (bridge). Các đồ thị con được tạo ra do loại bỏ đỉnh cắt hoặc cạnh cầu được gọi là khối (block) hoặc thành phần liên thông đôi (biconnected component).

a) Xác định tính liên thông

Ta hiệu chỉnh giải thuật **depthFirstSearch()** để xác định đồ thị có liên thông hay không. Bỏ dòng:

```

| while có một đỉnh v mà num(v) == 0
| 
```

Khi DFS() kết thúc, vì nó chỉ chạy trong một thành phần liên thông, nên nếu edges nhỏ hơn số cạnh hoặc i nhỏ hơn số đỉnh thì đồ thị không liên thông.

b) Xác định đỉnh khớp nối và các khối

Các đỉnh khớp nối có thể được phát hiện bằng cách mở rộng giải thuật DFS. Nhắc lại, khi duyệt DFS ta đánh số các đỉnh theo thứ tự duyệt, num(v) là thứ tự duyệt đỉnh v. num(v) < num(u) cho thấy v được duyệt trước u.

Ta định nghĩa pred(v): $\text{pred}(v) = \min(\text{pred}(v), \text{num}(u_1), \dots, \text{num}(u_k))$

Lúc đầu pred(v) khởi tạo bằng num(v), sau đó pred(v) được cập nhật dần với num(u_k) nhỏ nhất, u_k là đỉnh được duyệt trước v và kết nối với v hoặc con cháu trong nhánh DFS gốc v bằng một cạnh lùi. Nghĩa là, nếu nhánh DFS gốc v có nhiều cạnh lùi hướng lên phía gốc, ta ghi nhận cạnh lùi hướng lên cao nhất.

Một đỉnh v trong cây được gọi là đỉnh khớp nối khi:

- v là gốc của một nhánh DFS (cây tìm kiếm theo chiều sâu), và v có ít nhất một đỉnh con cháu trong nhánh DFS này.

- có ít nhất một cây con của v, có gốc là u, không chứa đỉnh nào kết nối với một trong những đỉnh trước v bằng một cạnh lùi. Nghĩa là nếu loại bỏ v thì từ u không có cách nào lên được đỉnh duyệt trước v. Điều kiện này xác định bởi $\text{pred}(u) \geq \text{num}(v)$.

blockDFS(v)

```

pred(v) = num(v) = i++;
for tất cả các đỉnh u kề với v
    if edge(vu) chưa được xử lý
        push(edge(vu));
    if num(u) == 0
        blockDFS(u);
    if pred(u) ≥ num(v)
        e = pop();
        while e != edge(vu)
            xuất e;
            e = pop();
        xuất e;
    
```

// đê quy đầu, xây dựng nhánh DFS gốc u, cây con của v
 // nếu không có cạnh từ u đến một đỉnh trên v, v là đỉnh khớp nối
 // xuất khôi bằng cách pop tất cả các cạnh trong stack
 // cho đến khi pop edge(vu) (cạnh duyệt đầu tiên của khôi)
 // e == edge(vu)

```

else pred(v) = min(pred(v), pred(u)); // ngược lại, nếu v không là đỉnh khớp nối, cập nhật pred(v)
else if u không là cha của v
    pred(v) = min(pred(v), num(u)); // nếu (vu) là cạnh lùi (u trước v), cập nhật pred(v)

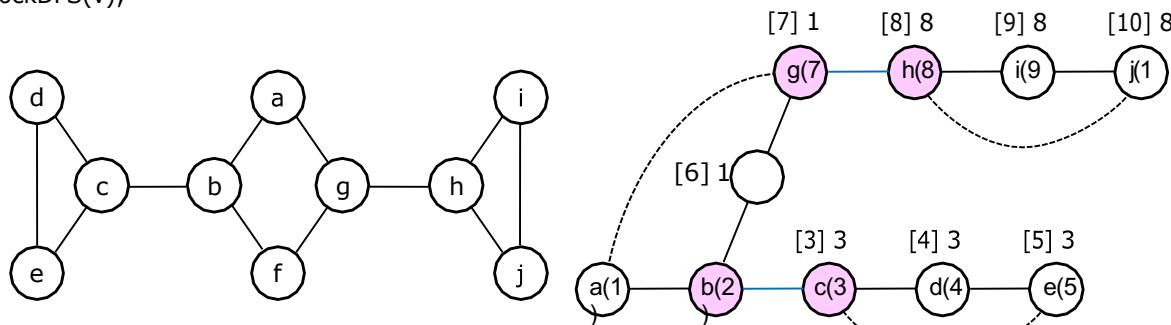
```

blockSearch()

```

for tất cả các đỉnh v
    num(v) = 0;
    i = 1;
while có một đỉnh v với num(v) == 0
    blockDFS(v);

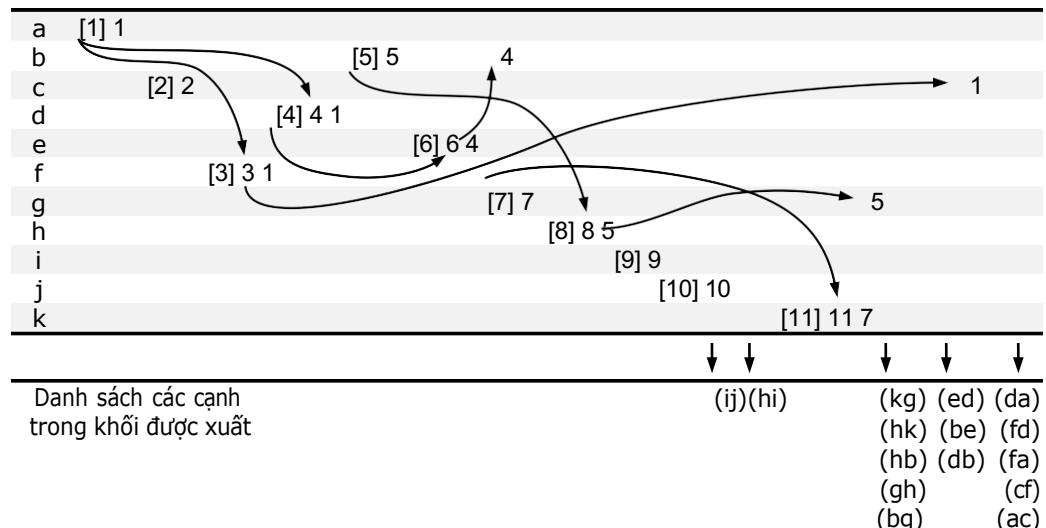
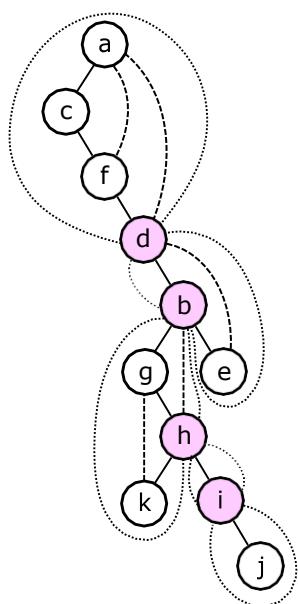
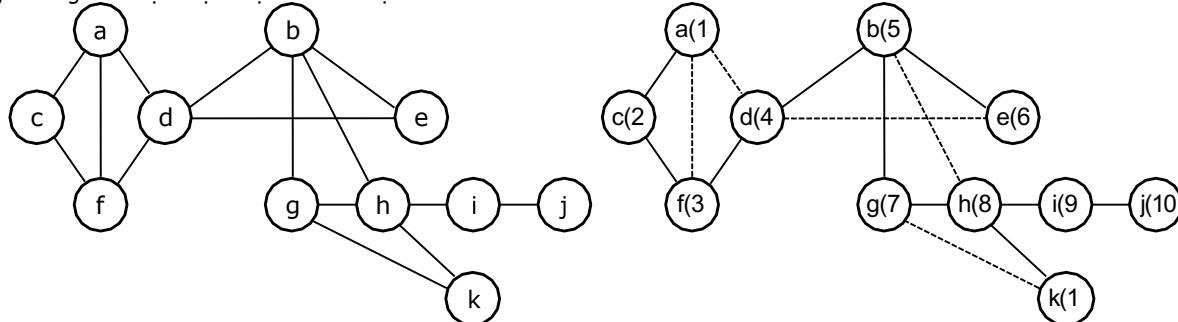
```



Để dễ hiểu giải thuật, chúng ta bắt đầu đơn giản hơn, bằng việc thử tìm các đỉnh khớp nối trong đồ thị trên, với kết quả duyệt DFS trong hình bên phải. Mỗi đỉnh v trong DFS được đánh nhãn $[x] y$, với $x = \text{num}(v)$ và $y = \text{pred}(v)$ đã tối ưu. DFS bắt đầu từ đỉnh a và duyệt đến e . Cạnh lùi (ec) được phát hiện và $\text{pred}(e) = \text{num}(c) = 3$. Sau đó, DFS quay lui về d , $\text{pred}(d) = \text{pred}(e) = 3$. Tương tự, khi DFS quay lui về c , $\text{pred}(c) = \text{pred}(d) = 3$. Bây giờ, do $\text{pred}(d) \geq \text{num}(c)$, đỉnh c được xem là đỉnh khớp nối. Khi DFS quay lui đến b , nó cũng thấy $\text{pred}(c) \geq \text{num}(b)$ và đỉnh b cũng là đỉnh khớp nối. Tại đỉnh b , DFS xuống nhánh mới đến đỉnh f rồi cuối cùng xuống đến j . Cạnh lùi (jh) được phát hiện và $\text{pred}(j) = \text{num}(h) = 8$. Khi DFS quay lui về i và h , $\text{pred}(h) = \text{pred}(i) = \text{pred}(j) = 8$. Do $\text{pred}(i) \geq \text{num}(h)$, đỉnh h là đỉnh khớp nối, cũng do $\text{pred}(h) \geq \text{num}(g)$, đỉnh g cũng là đỉnh khớp nối. Tại đỉnh g , cạnh lùi (ga) được phát hiện và $\text{pred}(g) = \text{num}(a) = 1$. Cuối cùng, $\text{pred}(b) = \text{pred}(f) = \text{pred}(g) = 1$ và DFS kết thúc tại đỉnh a . Trong cây DFS (không tính cạnh lùi), bậc của a không lớn hơn 1 nên a không được xem là đỉnh khớp nối.

Trong lúc duyệt DFS, các cạnh được đưa vào một stack, khi phát hiện đỉnh khớp nối, các cạnh thuộc khối được đẩy ra khỏi stack. Chú ý là do DFS dùng đệ quy đầu nên khối có đỉnh khớp nối được phát hiện cuối cùng sẽ được xuất ra đầu tiên.

Ta cũng xác định được các cạnh cầu (vu) khi $\text{pred}(u) \geq \text{num}(u)$. Trong ví dụ trên, u là các đỉnh c và h , các cạnh (bc) và (gh) là cạnh cầu. Bây giờ, xem giải thuật thực hiện với đồ thị sau:

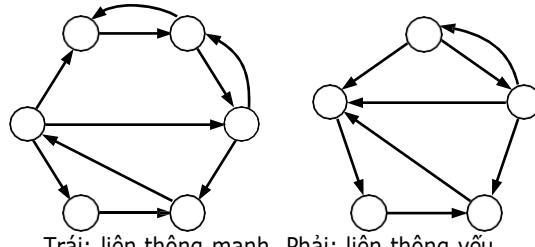


Trong bảng liệt kê, ta thấy $\text{blockDFS}(v)$ xử lý một đỉnh v : $\text{num}(v)$ trong cặp [] ở đầu và dãy các $\text{pred}(v)$ được cập nhật tiếp sau. Mũi tên cho thấy việc cập nhật $\text{pred}(v) = \min(\text{pred}(v), \text{num}(u))$ khi tìm thấy cạnh lùi (vu) , u được duyệt trước v . Ví dụ: với đỉnh a , khởi tạo $\text{num}(a) = \text{pred}(a) = 1$, cạnh (ac) được đưa vào stack; $\text{num}(c) = \text{pred}(c) = 2$, cạnh (cf) được đưa tiếp vào stack; $\text{num}(f) = \text{pred}(f) = 3$. Sau đó, xét a như là hậu duệ của f , cạnh (fa) được đưa tiếp vào stack. Lúc này do $\text{num}(a) \neq 0$, ta đang xét một cạnh lùi, cập nhật $\text{pred}(f) = \min(\text{pred}(f), \text{num}(a)) = \min(3, 1) = 1$. Tiếp theo, $\text{num}(d) = \text{pred}(d) = 4$, cạnh (fd) được đưa tiếp vào stack. Sau đó, xét a như là hậu duệ của d , cạnh (da) được đưa tiếp vào stack. Lúc này do $\text{num}(a) \neq 0$, ta đang xét một cạnh lùi, cập nhật $\text{pred}(d) = \min(\text{pred}(d), \text{num}(a)) = \min(4, 1) = 1$. Cứ tiếp tục như thế đến khi DFS kết thúc. Theo điều kiện $\text{pred}(u) \geq \text{num}(v)$ và v có bậc lớn hơn 1 trong cây DFS, ta có các đỉnh d, b, h, i là các đỉnh khớp nối. a không là đỉnh khớp nối vì bậc của nó không lớn hơn 1 trong cây DFS.

Các cạnh đã xử lý được đưa vào stack, khi đỉnh khớp nối cuối cùng được phát hiện, các cạnh của khối có liên quan đến đỉnh khớp nối đó được xuất. Cứ như thế, các khối được xuất theo thứ tự từ trái sang phải trong bảng liệt kê trên.

2. Tính liên thông trong đồ thị có hướng

Đồ thị có hướng được gọi là *liên thông yếu* (weakly connected) nếu đồ thị vô hướng tương ứng là liên thông. Đồ thị có hướng được gọi là *liên thông mạnh* (strongly connected) nếu với mỗi cặp đỉnh u và v luôn tìm được hai đường đi: đường đi có hướng từ đỉnh u đến đỉnh v và đường đi có hướng từ đỉnh v đến đỉnh u .



Trái: liên thông mạnh. Phải: liên thông yếu.

Đồ thị có hướng có thể tổ hợp từ các thành phần liên thông mạnh (SCC - Strongly Connected Component), các thành phần này được định nghĩa như những tập đỉnh con của đồ thị. Trong đồ thị có hướng, ta quan tâm tìm các thành phần liên thông mạnh. Để xác định SCC, ta cũng dựa trên DFS. Gọi v là đỉnh đầu tiên của SCC tìm được bằng DFS, v được gọi là gốc của SCC. Như vậy, mỗi đỉnh u trong SCC đều đến được từ v , $\text{num}(v) < \text{num}(u)$, và chỉ sau khi các đỉnh như u được thăm, DFS mới *quay ngược lại* v . Ta nhận biết sự kiện này bằng $\text{pred}(v) = \text{num}(v)$, lúc này có thể xuất SCC gốc v tìm được.

Bài toán trở thành tìm các đỉnh v , gốc của các SCC, tương tự như tìm các điểm khớp nối trong đồ thị vô hướng. Ta cũng dùng $\text{pred}(v)$, $\text{pred}(v)$ là *số nhỏ nhất* được chọn từ $\text{num}(v)$ và $\text{pred}(u)$, u là đỉnh có thể đi đến từ v và thuộc cùng SCC với v .

Trước hết ta khởi gán cho $\text{pred}(v)$: $\text{pred}(v) = \text{num}(v)$ (v có cung đến chính v). Sau đó duyệt tất cả những đỉnh u nối từ v :

- Nếu u đã thăm (có trong stack) thì ta cập nhật $\text{pred}(v)$: $\text{pred}(v) = \min(\text{pred}(v), \text{num}(u))$.

- Nếu u chưa thăm thì ta gọi đệ quy để thăm u , sau đó cập nhật $\text{pred}(v)$: $\text{pred}(v) = \min(\text{pred}(v), \text{pred}(u))$

Khi duyệt xong. Ta so sánh $\text{pred}(v)$ và $\text{num}(v)$. Nếu như $\text{pred}(v) = \text{num}(v)$ thì v đúng là gốc của SCC, do không có cung nối từ một đỉnh thuộc nhánh DFS gốc v tới một đỉnh được thăm trước v , nên cập nhật như thế nào $\text{pred}(v)$ cũng không nhỏ hơn $\text{num}(v)$.

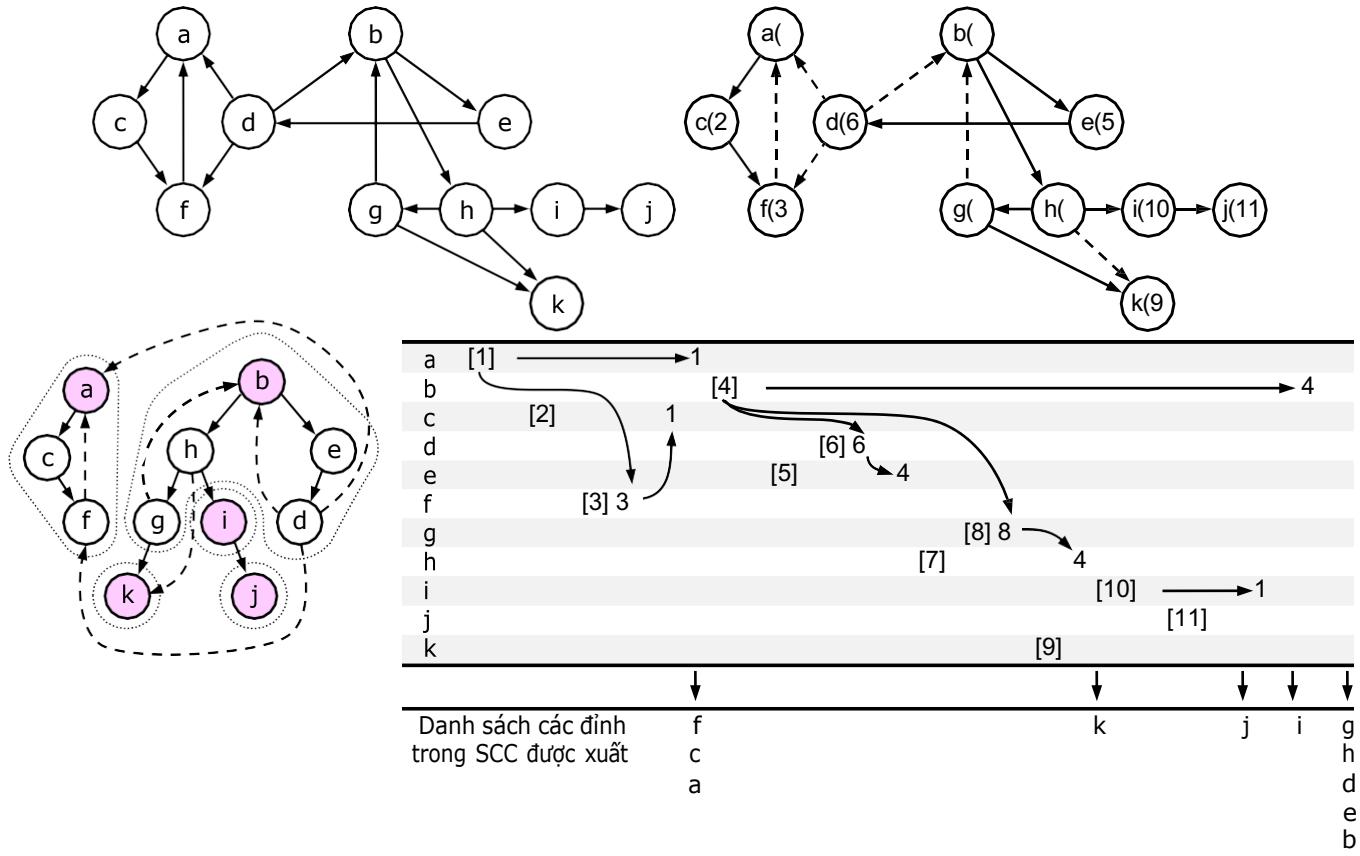
Giải thuật Tarjan tìm thành phần liên thông mạnh:

```
strongDFS(v)
    prev(v) = num(v) = i++;
    push(v);
    for tất cả các đỉnh u kề với v
        if num(u) == 0
            strongDFS(u);                                // đệ quy đầu để duyệt SCC
            pred(v) = min(pred(v), pred(u));             // backtrack để cập nhật pred
        else if num(u) < num(v) and u có trong stack
            pred(v) = min(pred(v), num(u));              // cập nhật pred nếu thấy cạnh lùi đến đỉnh u trong cùng SCC
        if pred(v) == num(v)                            // xác định v là gốc của SCC
            w = pop();                                 // xuất các đỉnh của SCC từ stack, cho đến khi gặp lại đỉnh v
            while w != v
                xuất w;
                w = pop();
            xuất w;
```

```
stronglyConnectedComponentSearch()
```

```
for tất cả các đỉnh v
    num(v) = 0;
    i = 1;
    while có một đỉnh v sao cho num(v) == 0          // duyệt tất cả các SCC
        strongDFS(v);                                // duyệt một SCC
```

Đồ thị có hướng được xử lý bởi chuỗi lời gọi $\text{strongDFS}()$, gán các đỉnh từ a đến k các số trong cặp () như hình trên phải. Trong lúc xử lý, năm SCC được tìm ra: $\{a,c,f\}$, $\{k\}$, $\{i\}$, $\{j\}$, $\{b,e,d,h,g\}$. Trong bảng liệt kê ta thấy $\text{strongDFS}(v)$ xử lý một đỉnh v : $\text{num}(v)$ trong cặp [] ở đầu và tiếp sau là dãy các $\text{pred}(v)$ được cập nhật. Nó cũng cho thấy các đỉnh của các SCC được xuất.



VII. Đồ thị Eulerian và Hamiltonian

1. Đồ thị Eulerian

Đường đi qua **tất cả các cạnh**, mỗi cạnh chỉ qua đúng một lần gọi là đường đi Eulerian. Đồ thị chứa đường đi Eulerian có hai đỉnh bậc lẻ. Đường đi Eulerian có đỉnh đầu và đỉnh cuối trùng nhau gọi là chu trình Eulerian. Đồ thị có chu trình Eulerian gọi là đồ thị Eulerian. Đồ thị Eulerian chỉ chứa các đỉnh có bậc chẵn.

a) Giải thuật Fleury

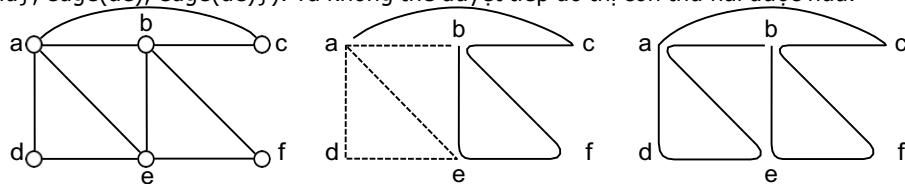
Chọn liên tiếp các cạnh chưa được duyệt, chỉ chọn cạnh là cầu (cạnh "một đi không trở lại") khi không còn cách chọn cạnh không là cầu. Nếu cạnh nào được chọn thì ghi nhận vào kết quả rồi loại bỏ cạnh đó trong tập các cạnh chưa được duyệt (untraversed).

fleuryAlgorithm(đồ thị VÔ hướng graph)

```

v = đỉnh bắt đầu (bất kỳ);
path = v;
untraversed = graph;
while v có một cạnh vu chưa duyệt
    if edge(vu) là cạnh chưa duyệt duy nhất
        e = edge(vu);
        loại v khỏi untraversed;
    else
        e = edge(vu) với vu không phải là cạnh cầu trong untraversed;
        path = path + u;
        loại e khỏi untraversed;
        v = u;
    if untraversed không còn cạnh
        có chu trình Eulerian;
    else không có chu trình Eulerian;
  
```

Trong ví dụ sau, bắt đầu duyệt tại đỉnh b. Giả sử ta đã duyệt đường befbcda đến đỉnh a. Bây giờ, ta có ba lựa chọn: edge(ab), edge(ad) và edge(ae). Nếu ta chọn **cạnh cầu** edge(ab), việc loại bỏ cạnh ab sẽ làm untraversed tách thành hai đồ thị con: (<{b}, Ø>) và (<{a,d,e}, {edge(ad), edge(ae), edge(de)})}. Ta không thể duyệt tiếp đồ thị con thứ hai được nữa.



Tìm chu trình Eulerian theo giải thuật Fleury. Phần đồ thị chưa duyệt (untraversed) là đường có nét đứt.

b) Bài toán "Người đưa thư Trung Quốc" (Chinese Postman problem)

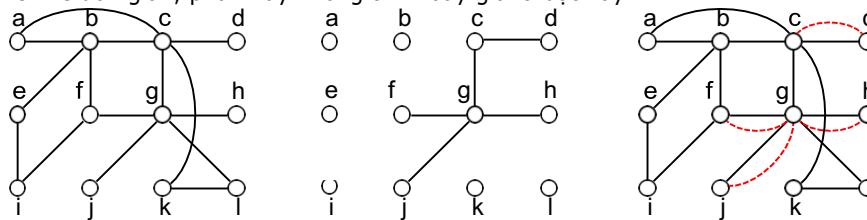
Phát biểu: người đưa thư lấy thư tại bưu cục, phân phôi thư đọc theo tất cả các con đường trong khu vực, rồi quay lại bưu cục. Xác định đường đi ngắn nhất cho người đưa thư.

Đường đi của người đưa thư được mô hình hóa thành đồ thị: cạnh là các con đường trong khu vực với chiều dài là trọng số, đỉnh là các ngã ba, ngã tư đường, ... Đường đi sẽ ngắn nhất nếu mỗi con đường chỉ đi qua một lần. Như vậy, nếu đồ thị là Eulerian thì mỗi chu trình Eulerian trong đồ thị là lời giải.

Nếu đồ thị không Eulerian, có thể tăng cường đồ thị để nó trở thành Eulerian bằng cách thêm vào một số cạnh:

- Cạnh thêm vào là lặp lại các cạnh có sẵn, liên thuộc với các đỉnh bậc lẻ, để đồ thị trở thành Eulerian.

- Tổng chiều dài các cạnh thêm vào là nhỏ nhất, xác định bằng giải thuật tìm bộ ghép tối thiểu (minimum matching) trên đồ thị con chứa các đỉnh bậc lẻ. Để đơn giản, phần này không trình bày giải thuật này.



Trái: đồ thị đường đi người đưa thư, có 6 đỉnh bậc lẻ.

Giữa: đồ thị các đỉnh bậc lẻ $\text{ODD} = \{c, d, f, g, h, j\}$

Phải: Các cạnh bổ sung để đồ thị thành Eulerian.

Như vậy, bài toán chuyển thành vấn đề nhóm các cạnh có đỉnh bậc lẻ sao cho tổng chiều dài các cạnh thêm vào là nhỏ nhất.

2. Đồ thị Hamiltonian

Chu trình Hamiltonian của một đồ thị là chu trình đi qua **tất cả các đỉnh** của đồ thị, mỗi đỉnh chỉ qua một lần (trừ đỉnh đầu và đỉnh cuối). Đồ thị chứa ít nhất một chu trình Hamiltonian gọi là đồ thị Hamiltonian. Không có tính chất đặc biệt nào dùng để nhận biết một đồ thị Hamiltonian. Tuy nhiên, mọi **đồ thị đầy đủ** là Hamiltonian.

a) Định lý Bondy-Chvátal

Định lý Bondy and Chvátal (1972); là sự mở rộng của định lý Dirak (1952) và định lý Ore (1960)

Nếu $\text{edge}(vu) \notin E$, đồ thị $G' = (V, E \cup \{\text{edge}(vu)\})$ là Hamiltonian, và $\deg(v) + \deg(u) \geq |V|$, thì đồ thị $G(V, E)$ là Hamiltonian.

Phát biểu khác: cho đồ thị $G(V, E)$, thêm một cạnh $\text{edge}(vu)$ vào đồ thị sao cho $\deg(v) + \deg(u) \geq |V|$, ta nhận được G' . Nếu G' là Hamiltonian thì G cũng là Hamiltonian.

Định lý trên cho phép từ một đồ thị Hamilton, loại bỏ một số cạnh để tạo đồ thị Hamilton khác. Điều này dẫn đến giải thuật tìm chu trình Hamilton như sau:

hamiltonCycle(**đồ thị G = (V,E)**)

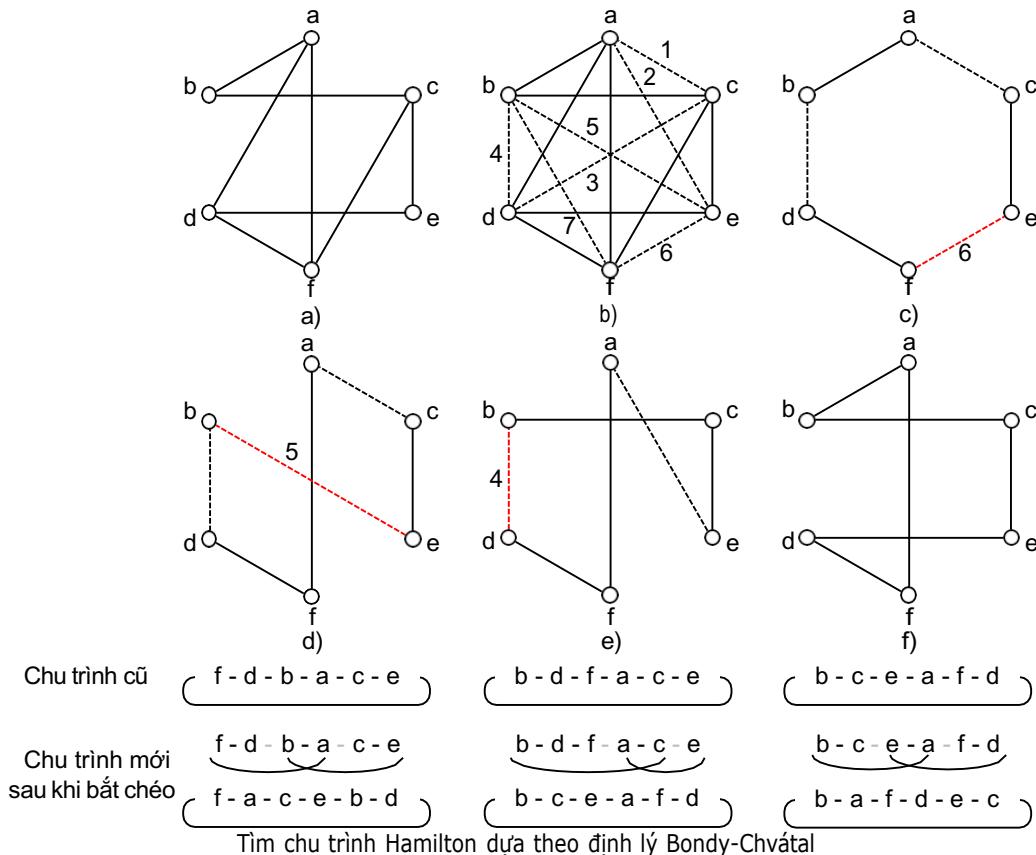
```

đặt nhãn cho tất cả các cạnh bằng 0;
k = 1;
H = E;
GH = G;
while GH chứa các đỉnh không kề v, u sao cho deg(v) + deg(u) ≥ |V|
    H = H ∪ {edge(vu)};
    GH = (V,H);
    label(edge(vu)) = k++;
if tồn tại một chu trình Hamiltonian C
    while (k = max{label(edge(pq)): edge(pq) ∈ C}) > 0
        C = chu trình tạo bằng cách bắt chéo với cạnh có nhãn < k;
```

Ví dụ minh họa cho đồ thị hình a). Xem trang sau.

Trong bước đầu, vòng lặp while thêm các cạnh để có đồ thị b). Cụ thể, đầu tiên với a chọn đỉnh không kề c, $\deg_H(a) + \deg_H(c) = 6 \geq |V| = 6$, thêm $\text{edge}(ac)$ với nhãn 1. Chú ý bậc của a thành 4, với a chọn đỉnh không kề khác e, $\deg_H(a) + \deg_H(e) = 6$, thêm $\text{edge}(ae)$ với nhãn 2. Tiếp theo, với b chọn các đỉnh không kề d, e và f rồi tiến hành tương tự như trên.

Trong bước thứ hai, tìm thấy một chu trình Hamiltonian: a, c, e, f, d, b. Cạnh có nhãn cao nhất trong chu trình là $\text{edge}(ef)$. Các đỉnh trong chu trình được sắp thứ tự sao cho các đỉnh của cạnh này ở hai đầu ngoài cùng. Rồi từ chuỗi đỉnh này, ta thử tìm hai cạnh bắt chéo bằng cách kiểm tra các cạnh nối chéo từ hai đỉnh lâng giềng trong với hai đầu. Ví dụ, trong lần thử đầu, ta chọn hai cạnh bắt chéo $\text{edge}(fb)$ và $\text{edge}(de)$ nhưng loại bỏ vì nhãn của $\text{edge}(fb)$ lớn hơn nhãn lớn nhất của chu trình là 6. Lần thử thứ hai thành công với $\text{edge}(fa)$ nhãn 0 và $\text{edge}(be)$ nhãn 5, chu trình trở thành f, a, c, e, b, d. Tiếp tục với cạnh có nhãn lớn nhất trong chu trình be. Cứ như vậy cho đến khi ta có chu trình với các cạnh **đều có nhãn 0**, nghĩa là chu trình chỉ chứa các cạnh từ G.



b) Bài toán "Người bán hàng" (Traveling Salesman problem)

Phát biểu: một người bán hàng đi vòng qua các thành phố, mỗi thành phố một lần, rồi quay lại thành phố xuất phát. Tìm hành trình ngắn nhất cho người bán hàng.

Đường đi của người bán hàng được mô hình hóa như một đồ thị có trọng số với đỉnh là thành phố và cạnh là đường nối giữa các thành phố. Bài toán chuyển thành vấn đề tìm chu trình Hamiltonian có tổng trọng số nhỏ nhất.

VIII. Tô màu đồ thị

Phát biểu: gán màu cho các đỉnh của đồ thị sao cho hai đỉnh của một cạnh có màu khác nhau và số màu sử dụng ít nhất.

Nhiều vấn đề thực tiễn được giải quyết bằng cách mô hình hóa thành bài toán tô màu đồ thị.

Ví dụ: với hình thức học chẽ tín chỉ, sinh viên có thể chủ động chọn đăng ký môn học theo kế hoạch học tập của mình. Điều này làm cho việc xếp lịch thi trở nên khó khăn. Lịch thi phải được xếp sao cho không có sinh viên nào thi nhiều hơn một môn tại cùng một thời điểm. Bài toán xếp lịch thi được mô hình hóa thành bài toán tô màu đồ thị như sau: lập đồ thị có các đỉnh là các môn thi, hai môn thi kề nhau nếu có một sinh viên thi cả hai môn này. Thời điểm thi của mỗi môn được biểu thị bằng các màu khác nhau.

Số màu tối thiểu được dùng để tô màu một đồ thị G gọi là sắc số (chromatic number) $\chi(G)$. Trong một số trường hợp đặc biệt, có thể xác định được sắc số của đồ thị: đồ thị đầy đủ $\chi(G_n) = n$, chu trình C_{2n} với số cạnh chẵn $\chi(C_{2n}) = 2$, chu trình C_{2n+1} với số cạnh lẻ $\chi(C_{2n+1}) = 3$, đồ thị G hai thành phần $\chi(G) \leq 2$.

Xác định sắc số của một đồ thị là vấn đề phức tạp (NP-completeness), tuy nhiên có nhiều cách cho phép tìm được số màu xấp xỉ sắc số, giúp tô màu đồ thị với số màu không lớn hơn sắc số nhiều quá.

1. Tô màu liên tiếp (sequential coloring)

Chuẩn bị chuỗi các đỉnh và chuỗi các màu với chỉ số tăng dần, tô đỉnh kế tiếp với chỉ số màu nhỏ nhất có thể.

sequentialColoringAlgorithm(graph = (V, E))

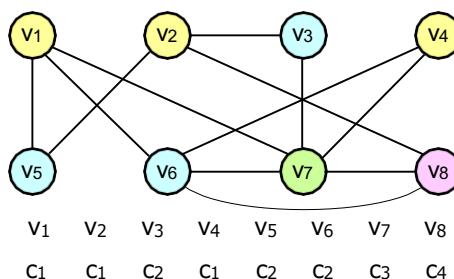
đặt các đỉnh theo thứ tự $v_1, \dots, v_{|V|}$;

đặt các màu theo thứ tự c_1, \dots, c_k ;

for $i = 1$ to $|V|$

$j =$ chỉ số màu nhỏ nhất không xuất hiện trong các đỉnh kề với v_i ;

$\text{color}(v_i) = c_j$;

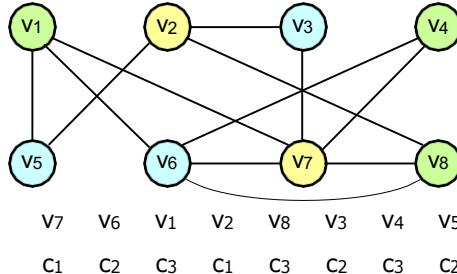


Welsh và Powell chứng minh rằng, trong giải thuật tô màu liên tiếp, số màu cần tô cho đồ thị là:

$$\chi(G) \leq \max_{i} \min(i, \deg(v_i) + 1)$$

Theo đó, để có $\chi(G)$ tối ưu, cần tổ chức sao cho các đỉnh có bậc cao nằm phía đầu danh sách các đỉnh. Ví dụ với đồ thị trên:

$\chi(G) \leq \max(\min(1, \deg(v_7) + 1), \min(2, \deg(v_6) + 1), \min(3, \deg(v_1) + 1), \min(4, \deg(v_2) + 1), \min(5, \deg(v_8) + 1), \min(6, \deg(v_3) + 1), \min(7, \deg(v_4) + 1), \min(8, \deg(v_5) + 1)) = \max(1, 2, 3, 4, 4, 3, 3, 3) = 4$.



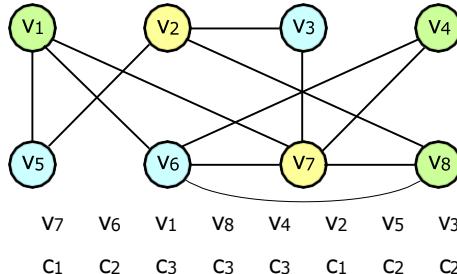
2. Giải thuật Brélaz

Khi chọn đỉnh để tô màu, ta có thể dùng nhiều điều kiện hơn. Trong giải thuật Brélaz, điều kiện thứ nhất là độ thâm màu (saturation degree) của đỉnh v , tức số màu khác nhau được tô cho các đỉnh kề của v .

brélazColoringAlgorithm(graph = (V, E))

```

for mỗi đỉnh v
    saturationDeg(v) = 0;
    uncoloredDeg(v) = degree(v);
    đặt các màu theo thứ tự  $c_1, \dots, c_k$ ;
    while các đỉnh chưa được xử lý xong
         $v =$  đỉnh có saturationDeg lớn nhất, hoặc nếu saturationDeg như nhau, là đỉnh có uncoloredDeg lớn nhất;
         $j =$  chỉ số màu nhỏ nhất không xuất hiện trong các đỉnh kề với  $v$ ;
        for mỗi đỉnh chưa tô u kề với  $v$ 
            if không có đỉnh kề với  $u$  (khác  $v$ ) được tô màu  $c_j$ 
                saturationDeg(u)++;
                uncoloredDeg(u)--;
            color(v) =  $c_j$ ;
```



Ta chọn đỉnh tô màu kề tiếp (trong các đỉnh còn lại) theo thứ tự ưu tiên: saturationDeg lớn nhất, uncoloredDeg lớn nhất, chỉ số thấp nhất.

Khi tô màu một đỉnh v , cặp (saturationDeg, uncoloredDeg) của các đỉnh kề sẽ thay đổi:

- saturationDeg tăng nếu không có đỉnh kề nào của nó (ngoại trừ v) có màu đang tô.
- uncoloredDeg giảm.

Đầu tiên v_7 được chọn và gán màu c_1 do v_7 có bậc cao nhất. Điều này làm thay đổi cặp (saturationDeg, uncoloredDeg) của các đỉnh kề đỉnh v_7 là $v_1 (1, 2)$, $v_3 (1, 1)$, $v_4 (1, 1)$, $v_6 (1, 3)$ và $v_8 (1, 2)$. Trong năm đỉnh có saturationDeg lớn nhất (1), ta chọn v_6 do có uncoloredDeg lớn nhất. Việc tô màu v_6 là thay đổi cặp (saturationDeg, uncoloredDeg) của v_1 là (2, 1) và v_8 (2, 1). Hai đỉnh ứng viên v_1 và v_8 có cặp (saturationDeg, uncoloredDeg) như nhau nên ta chọn v_1 do có chỉ số thấp nhất. Giải thuật tô màu cứ thế tiến hành.

Chú ý, trước khi tô màu đỉnh v_2 , lúc này cặp (saturationDeg, uncoloredDeg) của v_5 và v_3 đều là (1, 1). Khi tô màu v_2 với màu chọn c_1 , v_5 thay đổi thành (2, 0); tuy nhiên, do v_3 kề với v_7 đã tô màu c_1 nên saturationDeg của v_3 không tăng và v_3 thay đổi thành (1, 0). Vì vậy v_5 là đỉnh được chọn kế tiếp.

Vòng lặp while thực hiện với thời gian $|V|$, v được tìm trong $O(|V|)$ bước và vòng lặp for cần $\deg(v)$ bước, cũng $O(|V|)$. Vì vậy giải thuật chạy trong thời gian $O(|V|^2)$.

Sắp xếp

Hiệu quả việc xử lý dữ liệu, chẳng hạn như tìm kiếm, cơ bản sẽ tăng nếu dữ liệu được *sắp xếp trước* theo một tiêu chuẩn hay một thứ tự nào đó. Bước đầu tiên là chọn điều kiện sẽ được dùng để sắp xếp dữ liệu. Điều kiện này khác nhau tùy theo ứng dụng và được xác định bởi người dùng. Điều kiện có thể là sắp xếp tăng/giảm, sắp xếp theo thứ tự alphabet/dãy số nguyên, theo mã nhân viên, theo thu nhập, ... Sau khi chọn điều kiện, ta sẽ sắp xếp dữ liệu theo điều kiện đã chọn.

Có nhiều phương pháp sắp xếp khác nhau, nhưng chỉ có một số cách sắp xếp được xem là có ý nghĩa và hiệu quả. Ta phải xây dựng tiêu chuẩn đo lường tính hiệu quả và lựa chọn phương pháp so sánh thích hợp. Tiêu chuẩn đánh giá là các thuộc tính quan trọng của phương pháp sắp xếp, thường là số phép so sánh và số lần hoán chuyển dữ liệu.

I. Các giải thuật sắp xếp cơ bản

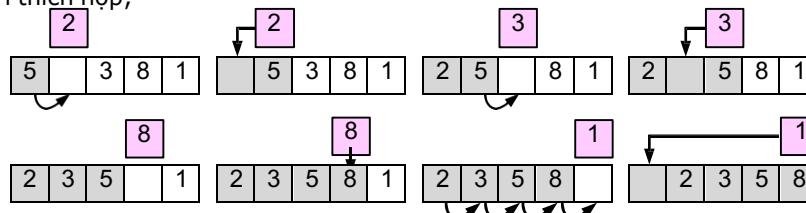
1. Sắp xếp chèn (Insert Sort)

Sắp xếp chèn dựa trên ý tưởng: chèn một phần tử chưa sắp xếp vào mảng đã được sắp xếp bằng cách tìm vị trí thích hợp cho nó và dịch chuyển các phần tử từ vị trí đó ra phía sau một vị trí, tạo vị trí trống để chèn phần tử đó vào.

Giải thuật bắt đầu với *mảng đã sắp xếp* có *một phần tử*, chính là phần tử đầu của mảng đầu vào. Vòng lặp chạy từ phần tử thứ hai trở về cuối, với mỗi phần tử, tìm vị trí thích hợp của nó trong *mảng đã sắp xếp* nằm ngay trước nó. Dịch chuyển các phần tử phía sau đi một vị trí và chèn phần tử đang xét vào.

insertSort(data[])

```
for (i = 1; i < data.length; i++)
    temp = data[i];
    chuyển tất cả các phần tử data[j] > temp ra sau một vị trí;
    đặt temp vào vị trí thích hợp;
```



Sắp xếp chèn mảng [5 2 3 8 1], ô trên mỗi mảng là temp, phần tô xám là mảng đã sắp xếp

Cài đặt:

```
void insertionsort(Comparable[] data) {
    Comparable temp;
    int i, j;
    for (i = 1; i < data.length; ++i) {
        temp = data[i];
        for (j = i; j > 0 && temp.compareTo(data[j - 1]) < 0; --j)
            data[j] = data[j - 1];
        data[j] = temp;
    }
}
```

Vòng lặp for bên trong có hai nhiệm vụ:

- tìm ra vị trí thích hợp cho temp.
- dịch chuyển các phần tử lớn hơn nó về phía sau.

Ưu điểm của giải thuật là *chỉ sắp xếp mảng khi cần thiết*, nếu mảng đã sắp xếp đúng thứ tự, không có phép hoán chuyển nào được thực hiện.

Khuyết điểm của giải thuật là nếu một phần tử được chèn vào vị trí mới, mọi phần tử lớn hơn nó trong mảng đã sắp xếp phải dịch chuyển về phía sau một phần tử, theo thứ tự từ sau ra trước.

Trường hợp tốt nhất, khi các phần tử đã đúng thứ tự. Tại mỗi vị trí i chỉ thực hiện 1 phép so sánh. Như vậy có $(n - 1)$ phép so sánh và $2(n - 1)$ phép gán (gán đến temp và gán từ temp). Thời gian thực hiện $O(n)$.

Trường hợp xấu nhất, khi các phần tử ban đầu có thứ tự đảo ngược:

- Tại bước thứ i của vòng lặp for bên ngoài, phải thực hiện i phép so sánh. Tổng số các phép so sánh là:

$$\sum_{i=1}^{n-1} i = 1 + 2 + \dots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

- Số lần gán để dịch chuyển trong vòng lặp for bên trong cũng tương tự: $O(n^2)$. Số lần gán cho temp và gán temp trở lại vị trí thích hợp là $2(n - 1)$. Tổng số các phép gán là:

$$\frac{n(n-1)}{2} + 2(n-1) = \frac{n^2 + 3n - 4}{2} = O(n^2)$$

Trường hợp các phần tử được phân bố ngẫu nhiên, ta xác định xem thời gian sắp xếp gần với trường hợp tốt nhất $O(n)$ hay trường hợp xấu nhất $O(n^2)$.

Ta nhận thấy nếu phần tử đang xét $data[i]$ khác j vị trí so với vị trí đúng của nó, thì $data[i]$ phải thực hiện phép so sánh với $(j + 1)$ phần tử khác. Do đó tại bước thứ i của vòng lặp ngoài, có 1, 2, ..., hoặc i phép so sánh. Giả sử các phần tử được lưu với xác suất như nhau. Số phép so sánh trung bình của $data[i]$ tại bước thứ i của vòng lặp ngoài là:

$$\frac{1+2+\dots+i}{i} = \frac{\frac{i(i+1)}{2}}{i} = \frac{i+1}{2}$$

Tổng các phép so sánh trong $(n - 1)$ lần thực hiện vòng lặp ngoài là:

$$\sum_{i=1}^{n-1} \frac{i+1}{2} = \frac{1}{2} \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} \frac{1}{2} = \frac{1}{2} \cdot \frac{n(n-1)}{2} + \frac{1}{2} (n-1) = \frac{n^2 + n - 2}{4}$$

Số phép so sánh gần bằng một nửa so với trường hợp xấu nhất, nhưng thời gian so sánh trung bình vẫn là $O(n^2)$.

Lập luận tương tự, tại bước thứ i của vòng lặp ngoài, số lần đổi chỗ của $data[i]$ hoặc là 0, 1, ... hoặc $i - 1$ lần. Số lần đổi chỗ trung bình là:

$$\frac{0+1+\dots+(i-1)}{i} = \frac{\frac{i(i-1)}{2}}{i} = \frac{i-1}{2}$$

Cộng với hai lần đổi chỗ cho biến $temp$ (gán cho $temp$ và gán $temp$ trở lại), Số lần đổi chỗ trung bình trong vòng lặp ngoài là:

$$\sum_{i=1}^{n-1} \left(\frac{i-1}{2} + 2 \right) = \frac{1}{2} \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} \frac{3}{2} = \frac{1}{2} \frac{n(n-1)}{2} + \frac{3}{2} (n-1) = \frac{n^2 + 5n - 6}{4}$$

Thời gian đổi chỗ trung bình là $O(n^2)$.

Kết quả cho thấy, số lần đổi chỗ và so sánh trong trường hợp ngẫu nhiên tương đương với trường hợp xấu nhất, là $O(n^2)$.

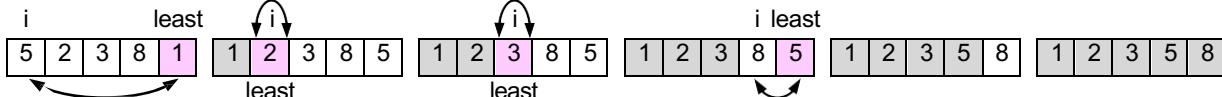
2. Sắp xếp chọn (Selection Sort)

Sắp xếp chọn dựa trên ý tưởng: chọn phần tử không đúng vị trí và đặt chúng vào vị trí thích hợp. Phương pháp này có giới hạn chế phép đổi chỗ các phần tử trong mảng.

Giải thuật sẽ tìm kiếm phần tử nhỏ nhất trong mảng và hoán chuyển nó với phần tử đầu tiên, như vậy phần tử đầu tiên đã đúng vị trí. Tiếp theo, giải thuật sẽ tìm kiếm phần tử nhỏ nhất trong *mảng còn lại* và hoán chuyển nó với phần tử đầu tiên của *mảng còn lại*. Nghĩa là, ở bước thứ i , giải thuật sẽ tìm kiếm phần tử nhỏ thứ i và hoán chuyển nó với phần tử thứ i của mảng. Lặp lại cho đến khi tất cả các phần tử đều được sắp xếp.

selectSort(data[])

```
for (i = 0; i < data.length - 1; i++)
    chọn phần tử nhỏ nhất trong data[i], ..., data[data.length - 1];
    hoán chuyển nó với data[i];
```



Sắp xếp chọn mảng [5 2 3 8 1], phần tô trắng là mảng còn lại chưa sắp xếp

Cài đặt:

```
void swap(Comparable[] data, int i, int j) {
    Comparable t = data[i]; data[i] = data[j]; data[j] = t;
}
```

```
void selectionsort(Comparable[] data) {
    int i, j, least; // least là chỉ số của phần tử nhỏ nhất
    for (i = 0; i < data.length - 1; ++i) {
        for (j = i + 1, least = i; j < data.length; ++j)
            if (data[j].compareTo(data[least]) < 0)
                least = j;
        if (least != i)
            swap(data, least, i);
    }
}
```

Vòng lặp for bên ngoài thực hiện $(n - 1)$ lần. Với mỗi bước i vòng lặp for bên trong thực hiện $(n - 1) - i$ lần. Số phép so sánh thực hiện trong vòng lặp trong là:

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + \dots + 1 = \frac{n(n-1)}{2} = O(n^2)$$

Số phép so sánh này *độc lập với mọi trạng thái của mảng*, do đó ta chỉ tối ưu được phép hoán chuyển.

Trường hợp tốt nhất, khi các phần tử đúng vị trí, không có phép hoán chuyển nào.

Trường hợp xấu nhất, số lần các phần tử trong vòng lặp đổi chỗ bằng số lần vòng lặp ngoài thực hiện, tại mỗi bước i , ta thực hiện ba phép gán để hoán chuyển. Số phép gán là $3(n - 1)$.

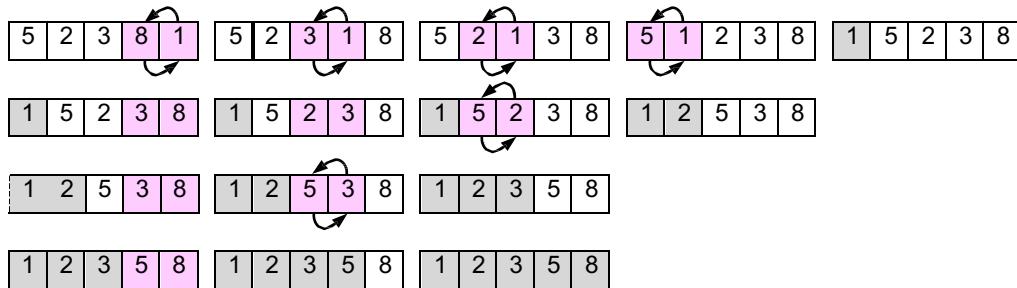
Ưu điểm của giải thuật này là số lượng các phép gán nhỏ hơn bất kỳ một giải thuật sắp xếp nào khác.

3. Sắp xếp nổi bọt (Bubble Sort)

Trong cách sắp xếp này, mảng được kiểm tra nhiều đợt, liên tiếp từ cuối mảng lên đầu mảng. Trong khi kiểm tra, nếu phần tử đang xét (j) và phần tử cạnh nó ($j - 1$) nghịch thế (sai chiều sắp xếp), chúng sẽ được hoán chuyển. Mỗi đợt kiểm tra thứ i , phần tử thứ i trong mảng sẽ "nổi bọt" lên phía đầu mảng đúng vị trí của nó.

bubbleSort(data[])

```
for (i = 0; i < data.length - 1; i++)
    for (j = data.length - 1; j > i; j--)
        đổi chỗ data[j] và data[j - 1] nếu nghịch thế
```



Sắp xếp nỗi bot mảng [5 2 3 8 1], mỗi hàng "nỗi bot" một phần tử, phần tô xám đã sắp xếp

Cài đặt:

```
void bubblesort(Comparable[] data) {  
    for (int i = 0; i < data.length - 1; ++i)  
        for (int j = data.length - 1; j > i; --j)  
            if (data[j].compareTo(data[j - 1]) < 0)  
                swap(data, j, j - 1);  
}
```

Chú ý, hai vòng lặp của sắp xếp nối ngược chiều nhau (ngoài tăng, trong giảm), vòng lặp bên ngoài là chấn trước của vòng lặp bên trong. Nếu bố trí ngược lại, ngoài giảm trong tăng, mảng được sắp xếp dần từ cuối mảng.

Số lăng so sánh là *giống nhau* trong các trường hợp (tốt, trung bình và xấu) và bằng tổng số lăng của vòng for bên trong:

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + \dots + 1 = \frac{n(n-1)}{2} = O(n^2)$$

Trường hợp xấu nhất, mảng sắp xếp theo thứ tự đảo ngược, công thức trên cũng được dùng để tính số lần hoán chuyển:

$$3 \frac{n(n-1)}{2}$$

Trường hợp tốt nhất, mảng đã đúng thứ tự, không có hoán chuyển.

Nếu các phần tử của mảng có thứ tự ngẫu nhiên, mảng được xử lý tại vòng lặp trong có $n - i$ phần tử (từ $\text{data}[i]$ đến $\text{data}[n - 1]$), số lần hoán chuyển của mảng con này là hoặc 0, hoặc 1, ..., hoặc $n - 1 - i$ lần. Số lần hoán chuyển trung bình là:

$$\frac{0+1+\dots+(n-1-i)}{n-i} = \frac{n-i-1}{2}$$

Tổng số lần thực hiện hoán chuyển của các vòng lắp trong là:

$$\sum_{i=0}^{n-1} i^2 = \frac{1}{2} \sum_{i=0}^{n-1} (n-1) - \frac{1}{2} \sum_{i=0}^{n-2} i = \frac{(n-1)^2}{2} - \frac{(n-1)(n-2)}{4} = \frac{n(n-1)}{4}$$

tương ứng với: $\frac{3}{4}n(n-1)$ phép gán.

Khuyết điểm chính của sắp xếp nổi bọt ở chỗ nó hoán chuyển hai phần tử nghịch thế một cách máy móc, nó không bỏ qua các phần tử đã đúng thứ tự mà hoán chuyển cả chúng.

Trong trường hợp trung bình, sắp xếp nối bot có số lần so sánh gấp đôi và số lần hoán chuyển tương đương với sắp xếp chèn; nó cũng có số lần so sánh tương đương và số lần hoán chuyển gấp n so với sắp xếp chọn.

Xem hoạt động một số phép so sánh tại: <http://www.cs.oswego.edu/~mohammad/classes/csc241/samples/sort/Sort2-E.html>

II. Các giải thuật sắp xếp hiệu quả

Các giải thuật sắp xếp cơ bản có thời gian thực hiện trung bình $O(n^2)$ là quá lớn; khi mảng tăng gấp đôi, thời gian sắp xếp tăng gấp bốn. Các giải thuật sắp xếp sau đều cố đạt mục tiêu là giảm thời gian thực hiện trung bình xuống $O(n \lg n)$.

1. Sắp xếp Shell (Shell Sort)

Giải thuật sắp xếp cơ bản sẽ có hiệu quả hơn nếu ta tiến hành sắp xếp thô mảng trước, rồi sắp xếp mịn dần. Nghĩa là ta có thể sắp xếp "nhảy cóc" các phần tử cách nhau h vị trí trên mảng trước, h giảm sau mỗi lần sắp xếp. Thứ tự các phần tử trong mảng nhờ đó được cải thiện dần, tiến gần đến trường hợp tốt nhất. Ý tưởng:

```
chia mảng data vào h mảng con  
for (i = 1; i ≤ h; i++)  
    sắp xếp mảng con data[i];  
    sắp xếp mảng data;
```

Ý tưởng này là nền tảng của phương pháp sắp xếp giảm dần trị tăng (diminishing increment sort) của Donald L. Shell:

- Chuẩn bị một dãy các trị tăng $h_1 = 1, h_2, \dots, h_t$. Các trị tăng h được dùng theo thứ tự ngược lại từ h_t đến $h_1 = 1$.

- Tại mỗi bước, mảng được chia thành h mảng con, và sắp xếp các mảng con này độc lập. Shell không chỉ ra phương pháp sắp xếp cụ thể nào cho mảng con, thông thường người ta dùng phương pháp sắp xếp chèn.

Thực chất, trong mỗi bước ta thực hiện sắp xếp chèn (so sánh, hoán chuyển) các phần tử cách nhau h vị trí trong mảng.

- Sau mỗi bước, trị h giảm dần cho đến $h_1 = 1$. Khi $h = 1$, giải thuật trở thành sắp xếp cơ bản một mảng (1-sort). Tuy nhiên, thứ tự sắp xếp trong mảng lúc này đã được cải thiện nhiều nên 1-sort thực hiện nhanh. Sắp xếp 1-sort, có thể dùng sắp xếp chèn hoặc sắp xếp nổi bọt.

shellSort(data[])

```
xác định dãy trị tăng  $h_1, \dots, h_t$  //  $h_1 = 1$ , dãy h sẽ được dùng từ sau ra trước  
for ( $h = h_t; t > 1; t--$ ,  $h = h_t$ )
```

chia mảng data vào h mảng con

```
for (i = 1; i <= h; i++)
    sắp xếp mảng con data[i];
```

sắp xếp mảng data;

data trước 5-sort	10	8	6	20	4	3	22	1	0	15	16
5 mảng con trước sắp xếp	10	-	-	-	-	3	-	-	-	-	16
	8	-	-	-	-	-	22				
		6	-	-	-	-	-	1			
			20	-	-	-	-	-	0		
				4	-	-	-	-	-	-	15
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----
5 mảng con sau sắp xếp	3	-	-	-	-	10	-	-	-	-	16
	8	-	-	-	-	-	22				
		1	-	-	-	-	-	6			
			0	-	-	-	-	-	20		
				4	-	-	-	-	-	-	15
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----
data trước 3-sort	3	8	1	0	4	10	22	6	20	15	16
3 mảng con trước sắp xếp	3	-	-	0	-	-	22	-	-	-	15
	8	-	-	-	4	-	-	6	-	-	16
		1	-	-	-	10	-	-	20		
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----
3 mảng con sau sắp xếp	0	-	-	3	-	-	15	-	-	22	
	4	-	-	-	6	-	-	8	-	-	16
		1	-	-	-	10	-	-	20		
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----
data trước 1-sort	0	4	1	3	6	10	15	8	20	22	16
data sau 1-sort	0	1	3	4	6	8	10	15	16	20	22

Sắp xếp Shell với $h = \{5, 3, 1\}$

Lựa chọn dãy h là yếu tố quyết định hiệu quả giải thuật. h quá nhỏ, các mảng con sẽ lớn và giải thuật sắp xếp sẽ không hiệu quả; h quá lớn, nhiều mảng con được tạo và dù chúng được sắp xếp, cơ bản chúng vẫn không cải thiện thứ tự của mảng ban đầu. Nghiên cứu lý thuyết cho thấy nên chọn dãy h như sau:

$h_1 = 1, h_{i+1} = 3h_i + 1$, chặn trên là h_t , sao cho $h_{t+2} \geq n$, n là kích thước của mảng.

Cài đặt:

```
void shellSort(Comparable[] data) {
    int i, j, k, h, hCnt, increments[] = new int[20];
    for (h = 1, i = 0; h < data.length; ++i) { // khởi tạo mảng các trị tăng
        increments[i] = h;
        h = 3 * h + 1;
    }
    for (i--; i >= 0; --i) { // sử dụng mảng h từ cuối lên đầu (phần tử đầu là 1)
        h = increments[i]; // lặp với h là phần tử nhỏ dần trong increments
        for (hCnt = h; hCnt < 2*h; ++hCnt) { // lặp trên số mảng con h-sort
            for (j = hCnt; j < data.length; ) { // sắp xếp chèn "nhảy cóc" h vị trí
                Comparable tmp = data[j];
                k = j;
                while (k-h >= 0 && tmp.compareTo(data[k-h]) < 0) {
                    data[k] = data[k - h];
                    k -= h;
                }
                data[k] = tmp;
                j += h;
            }
        }
    }
}
```

Trong trường hợp xấu nhất, số lần so sánh của sắp xếp Shell với $h_{i+1} = 3h_i + 1$ là $O(n^{3/2})$, tốt hơn $O(n^2)$ của sắp xếp chèn, nhưng vẫn lớn hơn độ phức tạp mong đợi là $O(n \lg n)$.

2. Sắp xếp dùng heap (Heap sort)

Phương pháp sắp xếp chọn cần thực hiện $O(n^2)$ phép so sánh, bù lại số phép hoán chuyển lại tương đối ít. Nếu ta có thể cải tiến giải thuật nhằm giảm bớt phép so sánh thì kết quả rất có triển vọng. John Williams đề xuất giải thuật sắp xếp dùng heap (còn gọi là sắp xếp vun đồng) có liên quan đến giải thuật sắp xếp chọn.

Giả sử ta muốn sắp xếp mảng theo thứ tự tăng dần:

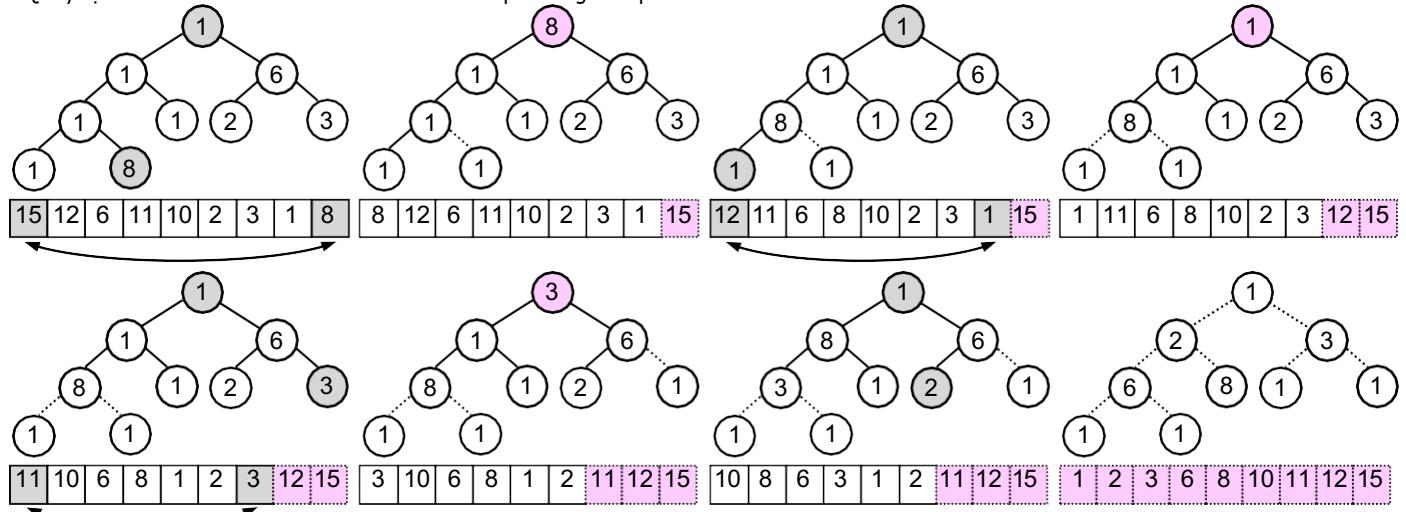
- Sắp xếp chọn "ngược", chọn phần tử lớn nhất đặt vào cuối mảng, chọn phần tử lớn thứ hai đặt vào vị trí thứ hai từ cuối mảng trở lên (phát biểu đệ quy: phần tử lớn thứ hai đặt vào cuối mảng còn lại), ... cứ thế cho đến khi còn 1 phần tử.

- Để chọn phần tử lớn nhất của một mảng, ta dùng cấu trúc heap đã trình bày trong phần cây nhị phân. Khi mảng được tổ chức thành maxheap, phần tử đầu của mảng (đỉnh heap) là phần tử lớn nhất.

Giải thuật sắp xếp dùng heap:

- Đầu tiên tổ chức mảng thành heap, ta có thể dùng phương pháp của Floyd (bottom-up).
- Hoán chuyển phần tử lớn nhất (phần tử đầu mảng) với phần tử cuối mảng.
- Tổ chức lại (còn gọi là vun đồng hay heapification) heap với mảng còn lại, không tính phần tử cuối đã sắp xếp.

- Quay lại bước 2 và tiến hành cho đến khi heap không còn phần tử nào.



Thực hiện sắp xếp với heap xây dựng từ mảng đầu vào. Các ô màu đỏ đã được sắp xếp, từ sau ra trước
7 hình đầu: sắp xếp được 3 phần tử. Hình cuối: sắp xếp hoàn tất

heapSort(data[])

```
chuyển data thành heap;
for (i = data.length - 1; i > 1; i--) {
    hoán chuyển data[0] với data[i] // 0 là vị trí đầu mảng, tức đỉnh heap; i là vị trí cuối của mảng đang xét
    phục hồi tính chất heap của mảng còn lại data[0], .. , data[i - 1].
```

Để phục hồi tính chất heap của mảng, ta dùng hàm moveDown() đã thảo luận trong phần heap. Hàm moveDown() cũng được dùng khi chuyển mảng thành heap. Chú ý, bản chất của giải thuật Floyd là thực hiện moveDown() với các phần tử từ node không phải lá cuối cùng ngược lên đầu mảng.

Cài đặt:

```
void moveDown(Comparable[] data, int first, int last) {
    int largest = 2 * first + 1;
    while (largest <= last) {
        if (largest < last && data[largest].compareTo(data[largest + 1]) < 0)
            largest++;
        if (data[first].compareTo(data[largest]) < 0) {
            swap(data, first, largest);
            first = largest;
            largest = 2 * first + 1;
        } else largest = last + 1;
    }
}

void heapSort(Comparable[] data) {
    for (int i = data.length / 2 - 1; i >= 0; --i)
        moveDown(data, i, data.length - 1);
    for (int i = data.length - 1; i >= 1; --i) {
        swap(data, 0, i);
        moveDown(data, 0, i - 1);
    }
}
```

Trong giai đoạn đầu tiên, giải thuật gọi moveDown() để tạo heap, thời gian thực hiện là $O(n)$.

Trong giai đoạn sắp xếp, giải thuật thực hiện $n - 1$ lần hoán chuyển giữa phần tử đỉnh heap và phần tử tại vị trí i . Mỗi bước đều phải phục hồi tính chất heap (vun đống) bằng moveDown(). Trong trường hợp xấu nhất, moveDown() lặp $\lg(i)$ lần để chuyển root xuống. Như vậy, tổng số lần thực hiện moveDown() trong bước thứ hai là: $\sum_{i=1}^{n-1} \lg i$ tương đương $O(n \lg n)$.

Trường hợp xấu nhất, giải thuật cần: $O(n)$ để tạo heap, $(n - 1)$ lần hoán chuyển và $O(n \lg n)$ cho phục hồi heap. Độ phức tạp là: $O(n) + O(n \lg n) + (n - 1) = O(n \lg n)$.

Trường hợp tốt nhất, khi mảng chứa các phần tử giống nhau. Giai đoạn đầu, moveDown() được gọi $n/2$ lần, nhưng không thực hiện phép chuyển nào. Trong giai đoạn hai, giải thuật thực hiện $(n - 1)$ lần hoán chuyển. Trong trường hợp tốt nhất, n phép so sánh được thực hiện trong giai đoạn đầu và $2(n - 1)$ phép so sánh cho giai đoạn 2. Như vậy, tổng các phép so sánh trong trường hợp tốt nhất là $O(n)$. Tuy nhiên, nếu mảng chứa các phần tử phân biệt, số phép so sánh bằng $n \lg n - O(n)$.

3. Sắp xếp nhanh (Quick Sort)

Được đề xuất bởi C.A.R. Hoare, quicksort dựa trên ý tưởng "chia để trị". Mảng ban đầu được chia thành hai mảng con: mảng thứ nhất chứa các phần tử nhỏ hơn hoặc bằng phần tử làm mốc (pivot, hay trị biên, bound), mảng thứ hai chứa các phần tử lớn hơn hoặc bằng phần tử làm mốc. Như vậy, việc sắp xếp mảng ban đầu được tiến hành bằng cách sắp xếp độc lập hai mảng con này. Việc sắp xếp các mảng con lại được tiến hành tương tự, nghĩa là chia chúng thành các mảng nhỏ hơn. Quá trình này lặp lại cho đến khi mảng con cần sắp xếp chỉ còn một phần tử.

Quicksort bản chất là một giải thuật đệ quy vì nó lặp đi lặp lại việc sắp xếp với các mảng con của nó.

quickSort(array [])

```
if length(array) > 1
    chọn trị biên bound;
    duyệt các phần tử el của mảng;
    cho el vào mảng subarray1 = {el: el ≤ bound};
    hoặc vào mảng subarray2 = {el: el ≥ bound};
    quickSort(subarray1);
    quickSort(subarray2);
```

Để phân hoạch một mảng, cần hai thao tác:

- Chọn trị làm mốc (pivot): có một số chiến lược chọn phần tử này, chọn phần tử đầu mảng hoặc giữa mảng. Ngoài việc chọn phần tử làm mốc, thủ tục này cần bảo đảm rằng trong quá trình phân hoạch mảng, vị trí của mốc không thay đổi.

Để làm điều này, trước tiên ta cất phần tử mốc vào vị trí đầu tiên. Sau khi thực hiện phân hoạch thành hai mảng con, chuyển phần tử mốc về vị trí thích hợp là vị trí cuối trong mảng con thứ nhất.

- Phân hoạch các phần tử của mảng vào hai mảng con:

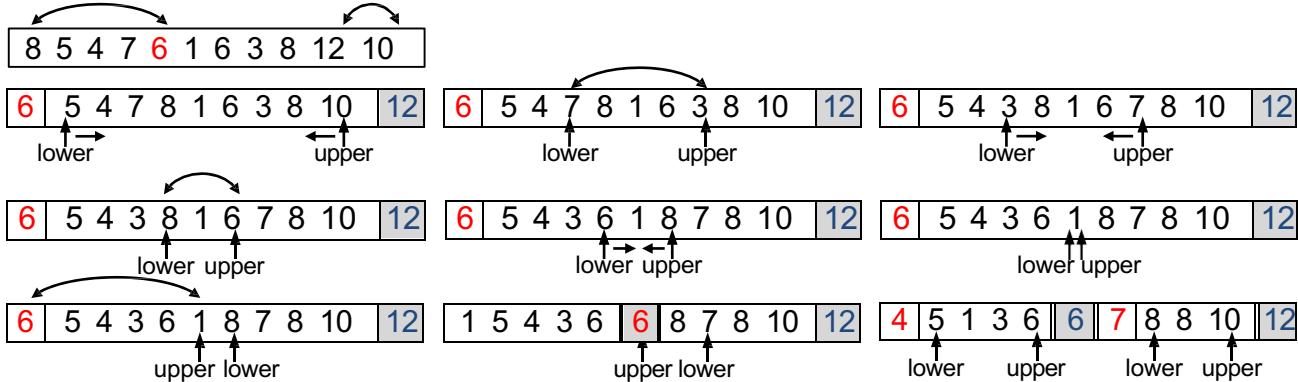
+ hai vòng lặp while bên trong dùng chọn phần tử vào hai mảng con: khi phần tử do lower chỉ đến còn nhỏ hơn trị mốc thì chọn nó (lower tăng) vào mảng con thứ nhất; khi phần tử do upper chỉ đến còn lớn hơn trị mốc thì chọn nó (upper giảm) vào mảng con thứ hai. Nếu cả hai con trỏ này dừng lại (thoát khỏi cả hai vòng lặp), hoán chuyển phần tử lớn hơn mốc ở phía đầu mảng (do lower chỉ đến) với phần tử nhỏ hơn mốc ở phía cuối mảng (do upper chỉ đến).

+ vòng lặp while ngoài cùng dùng kết thúc phân hoạch (lower > upper).

```
void quicksort(Comparable[] data, int first, int last) {
    int lower = first + 1, upper = last;
    swap(data, first, (first + last) / 2); // cất pivot
    Comparable bound = data[first];
    while (lower <= upper) {
        while (bound.compareTo(data[lower]) > 0)
            lower++;
        while (bound.compareTo(data[upper]) < 0)
            upper--;
        if (lower < upper) swap(data, lower++, upper--);
        else lower++;
    }
    swap(data, upper, first); // trả pivot về vị trí thích hợp, data[upper] đã được sắp xếp
    if (first < upper - 1) quicksort(data, first, upper - 1);
    if (upper + 1 < last) quicksort(data, upper + 1, last);
}
```

Chú ý, trong lần gọi đầu tiên (cho toàn mảng) nếu trị mốc tình cờ là phần tử lớn nhất của mảng, thì lower có thể tăng vượt quá giới hạn của mảng. Cách khắc phục đơn giản là xử lý trước, trong đó tìm phần tử lớn nhất và hoán chuyển nó xuống cuối mảng. Sau đó mới gọi phân hoạch đệ quy:

```
void quicksort(Comparable[] data) {
    if (data.length < 2) return;
    int max = 0;
    for (int i = 1; i < data.length; ++i) // tìm phần tử lớn nhất và đặt nó vào cuối mảng
        if (data[max].compareTo(data[i]) < 0) max = i;
    swap(data, data.length - 1, max);
    quicksort(data, 0, data.length - 2);
}
```



Tiền xử lý và bước phân hoạch đầu tiên của mảng [8 5 4 7 6 1 6 3 8 12 10]

Màu đỏ là trị pivot, màu xanh là trị chặn trên

Trường hợp xấu nhất, tại mỗi bước luôn chọn phải phần tử mốc là phần tử nhỏ nhất hoặc lớn nhất, một mảng con sẽ là mảng rỗng. Giải thuật thực hiện các thao tác trên các mảng có kích thước $n - 1, n - 2, n - 3, \dots, 2$. Các lần phân hoạch thực hiện số phép so sánh: $n - 2 + n - 3 + \dots + 1$, thời gian thực hiện là $O(n^2)$.

Trường hợp tốt nhất, khi phần tử mốc chia mảng thành hai mảng con có độ dài bằng nhau ($n/2$). Nếu các phần tử mốc của các mảng con cũng tốt như vậy, nghĩa là phần tử mốc luôn ở vị trí giữa của mảng, tổng số phép so sánh là:

$$n + 2 \frac{n}{2} + 4 \frac{n}{4} + \dots + n \frac{n}{n} = n(\lg n + 1)$$

Vậy thời gian thực hiện trong trường hợp tốt nhất là $O(n\lg n)$.

Các phân tích cho thấy trường hợp trung bình cần $O(n\lg n)$ phép so sánh, nghĩa là tương đương với trường hợp tốt nhất. Quicksort gần với sắp xếp lý tưởng, nó tốt hơn các phương pháp sắp xếp hiệu quả khác ít nhất hai lần.

Để tránh trường hợp xấu nhất, thủ tục phân hoạch phải sinh ra các mảng có kích thước xấp xỉ nhau, nghĩa là cần chọn phần tử mốc đủ tốt. Có hai phương pháp được thảo luận:

- Phương pháp sinh ngẫu nhiên: chọn một số ngẫu nhiên giữa first và last, số đó được dùng như chỉ số của phần tử mốc. Tuy nhiên, bộ sinh số ngẫu nhiên tốt có thể làm chậm thời gian thực hiện do chúng sử dụng các kỹ thuật phức tạp.

- Phương pháp chọn trung vị (median): ta chọn số có trị *năm* giữa trong ba phần tử (đầu, giữa, cuối) mảng. Ví dụ, trung vị của (260, 126, 305) là 260. Rõ ràng, xác suất để cả ba phần tử này cùng rất nhỏ hoặc cùng rất lớn trong mảng là bé.

Đối với mảng có kích thước nhỏ, quicksort có vẻ không thích hợp, nó kém hiệu quả hơn sắp xếp theo kiểu chèn. Tuy nhiên, đối với mảng có kích thước nhỏ, sai biệt về thời gian thực hiện không đáng kể.

4. Sắp xếp trộn (Merge Sort)

Trong trường hợp xấu nhất quicksort có độ phức tạp $O(n^2)$ do nó khó kiểm soát quá trình chia đôi mảng. Mặc dù cố gắng chọn phần tử mốc hợp lý, vẫn không có gì đảm bảo mảng được chia có kích thước xấp xỉ nhau.

Sắp xếp trộn dùng chiến lược chia đôi mảng đơn giản và tập trung vào việc trộn hai mảng con đã sắp xếp. Ý tưởng chính là trộn hai mảng con đã được sắp xếp thành một mảng có thứ tự. Chú ý, hai mảng con đã được sắp xếp này là kết quả của việc trộn hai nửa mảng con đã được sắp xếp trước đó.

Trước tiên, việc gọi đệ quy (rẽ nhánh) sẽ chia mảng ban đầu thành hai mảng con, rồi cứ thế chia nhỏ cho đến khi mảng con có ít hơn hai phần tử:

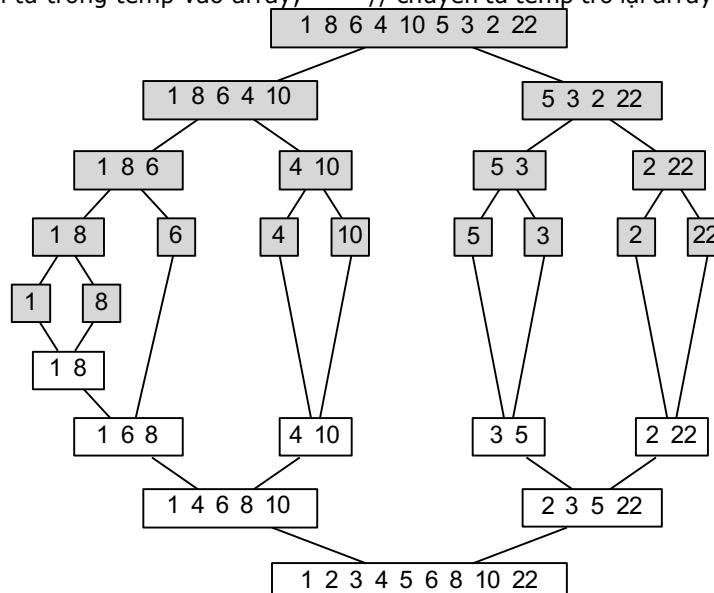
mergeSort(data)

```
if data có ít nhất hai phần tử
    mergeSort(nửa trái của data);
    mergeSort(nửa phải của data);
    trộn hai nửa thành mảng đã sắp xếp;
```

Sau khi kết thúc gọi đệ quy, quá trình trộn bắt đầu. Hai mảng con *đã sắp xếp* được trộn thành mảng có thứ tự. Tác vụ đơn giản này được thực hiện như sau: liên tiếp chọn phần tử nhỏ hơn từ đầu một trong hai phần còn lại của hai nửa mảng, đưa vào một mảng trung gian rồi sau đó sao chép kết quả vào mảng đích.

merge(array, first, last)

```
mid = (first + last) / 2;
i1 = 0;                                // chỉ số của mảng tạm temp
i2 = first;                             // chỉ số của mảng bên trái
i3 = mid + 1;                           // chỉ số của mảng bên phải
while hai nửa mảng trái và phải của array còn phần tử
    if array[i2] < array[i3]
        temp[i1++] = array[i2++];
    else
        temp[i1++] = array[i3++];
chuyển các phần tử còn lại trong array vào temp; // chỉ còn một mảng (trái hoặc phải)
chuyển tất cả các phần tử trong temp vào array; // chuyển từ temp trở lại array
```



Sắp xếp trộn [1 8 6 4 10 5 3 2 22]. Màu xám là gọi đệ quy.

```
void merge(Comparable[] array, int first, int last) {
    Comparable[] temp = new Comparable[last - first + 1];
    int mid = (first + last) / 2;
    int i1 = 0, i2 = first, i3 = mid + 1;
    while (i2 <= mid && i3 <= last)
```

```

if (array[i2].compareTo(array[i3]) < 0) temp[i1++] = array[i2++];
else temp[i1++] = array[i3++];
while (i2 <= mid) temp[i1++] = array[i2++];
while (i3 <= last) temp[i1++] = array[i3++];
for (i1 = 0, i2 = first; i2 <= last; array[i2++] = temp[i1++]) { }
}

void mergesort(Comparable[] data, int first, int last) {
int mid = (first + last) / 2;
if (first < mid) mergesort(data, first, mid);
if (mid + 1 < last) mergesort(data, mid + 1, last);
merge(data, first, last);
}

```

Do toàn bộ mảng array được sao chép đến mảng trung gian temp rồi lại được sao chép ngược trở lại, nên số lần đổi chỗ trong mỗi lần gọi merge() luôn bằng nhau và bằng $2(\text{last} - \text{first} + 1)$. Đổi với mảng n phần tử, số lần đổi chỗ được tính bằng công thức truy hồi sau:

$$\begin{cases} M(1)=0 \\ M(n)=2M\left(\frac{n}{2}\right)+2n \end{cases}$$

$M(n)$ có thể tính theo cách sau:

$$\begin{aligned} M(n) &= 2\left(2M\left(\frac{n}{4}\right)+2\left(\frac{n}{2}\right)\right)+2n = 4M\left(\frac{n}{4}\right)+4n \\ &= 4\left(2M\left(\frac{n}{8}\right)+2\left(\frac{n}{4}\right)\right)+4n = 8M\left(\frac{n}{8}\right)+6n \\ &\dots \\ &= 2^i M\left(\frac{n}{2^i}\right)+2in \end{aligned}$$

Chọn $i = \lg n$, tức $n = 2^i$, ta có số lần đổi chỗ: $M(n) = 2^i M\left(\frac{n}{2^i}\right) + 2in = nM(1) + 2n\lg n = 2n\lg n = O(n\lg n)$

Số lần so sánh trong trường hợp xấu nhất cũng tính theo công thức truy hồi:

$$\begin{cases} C(1)=0 \\ C(n)=2C\left(\frac{n}{2}\right)+n-1 \end{cases}$$

và cũng cho kết quả $O(n\lg n)$.

Sắp xếp trộn sẽ hiệu quả hơn nếu ta khử đệ quy và thay bằng vòng lặp. Tuy nhiên sắp xếp trộn có khuyết điểm nghiêm trọng là cần thêm vùng lưu trữ cho việc trộn các mảng.

5. Sắp xếp theo cơ sở (Radix Sort)

Khi sắp xếp, các số được lần lượt phân phôi vào các cột từ 0 đến 9 theo hàng số ngoài cùng bên phải (hàng đơn vị). Ví dụ 197 có hàng số ngoài cùng bên phải là 7 sẽ được xếp vào cột 7. Quá trình này lặp lại bằng cách phân phôi theo hàng số kế tiếp về bên trái (hàng chục), trường hợp này là 9. Quá trình này lặp lại cho đến hàng số cuối cùng có thể (hàng trăm, hàng ngàn, ...). Giải thuật này được tổng hợp lại trong đoạn mã giả sau:

radixSort()

```

for (d = 1; d <= vị trí của hàng số trái nhất của số dài nhất; d++)
    phân phôi các số vào các cột số từ 0 đến 9 theo hàng số thứ d;
    đặt các số vào một danh sách;

```

Điều quan trọng để đạt được chuỗi số mới thích hợp hơn là phân phôi số vào 10 cột rồi gom lại.

Tổ chức các cột có cấu trúc queue. Ví dụ 93, 63, 64 và 94 sẽ được chọn theo hàng đơn vị và phân phôi vào các cột 3 và 4 (các cột khác trống):

Cột 3: 93, 63

Cột 4: 64, 94

Các cột trên được gom lại tạo thành danh sách mới: 93, 63, 64, 94. Phân phôi tiếp theo hàng chục, các cột như sau:

Cột 6: 63, 64

Cột 9: 93, 94

Gom lại thành danh sách: 63, 64, 93, 94. Phân phôi tiếp theo hàng trăm:

Cột 0: 63, 64, 93, 94

Gom lại thành danh sách kết quả đã được sắp xếp: 63, 64, 93, 94.

Ta nhận thấy, quan hệ thứ tự của các phần tử trong cột được giữ lại. Khi các số được phân phôi vào cột theo hàng d, thì bên trong mỗi cột, các số đã được sắp xếp theo hàng trước đó ($d - 1$). Ví dụ sau bước 3, phân phôi theo hàng trăm, cột 5 chứa các số nguyên 12534, 554, 4590, thì cột này đã được sắp thứ tự theo hàng chục 12534, 554, 4590.

Cài đặt:

```

static int RADIX = 10;
void radixsort(int[] data) {
    int d, j, k, factor, digits;
    Queue<Integer>[] queues = new LinkedList[RADIX];
    for (d = 0; d < RADIX; ++d)

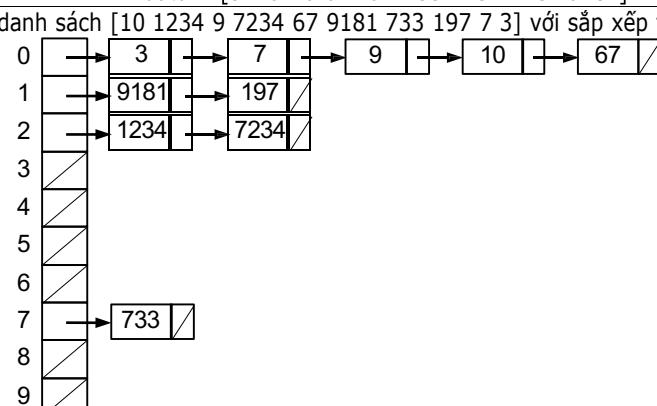
```

```

queues[d] = new LinkedList<>();
for (digits = factor = 1; digits != 0; factor *= RADIX) {
    for (j = 0; j < data.length; ++j)
        queues[(data[j] / factor) % RADIX].add(data[j]);
    for (j = k = 0; j < RADIX; ++j) {
        while (!queues[j].isEmpty()) { // gom các queue trở lại mảng data
            data[k++] = queues[j].remove();
            digits = j; // nếu chỉ còn queues[0] không rỗng, dùng giải thuật
        }
    }
}
}

```

Bước 1	data = [10 1234 9 7234 67 9181 733 197 7 3]									
cột:	0	1	2	3	4	5	6	7	8	9
	10	9181		3	7234		197	7		
			733	1234		67				9
Bước 2	data = [10 9181 733 3 1234 7234 67 197 7 9]									
cột:	0	1	2	3	4	5	6	7	8	9
	9		7234							
	7		1234							
	3	10		733		67				
Bước 3	data = [3 7 9 10 733 1234 7234 67 9181 197]									
cột:	0	1	2	3	4	5	6	7	8	9
	67									
	10									
	9									
	7	197	7234							
	3	9181	1234							
Bước 4	data = [3 7 9 10 67 9181 197 1234 7234 733]									
cột:	0	1	2	3	4	5	6	7	8	9
	733									
	197									
	67									
	10									
	9									
	7									
	3	1234								
							7234			9181
Sắp xếp danh sách [10 1234 9 7234 67 9181 733 197 7 3] với sắp xếp theo cơ số	data = [3 7 9 10 67 197 733 1234 7234 9181]									



Mảng các queue được sử dụng cho sắp xếp theo cơ số (bước 3, ví dụ trên)

Không giống như các giải thuật sắp xếp trước, giải thuật này dựa trên bản chất nội tại của phần tử, không dựa trên so sánh giữa các phần tử. Với mỗi số trong data, hai tác vụ được thực hiện để tập trung vào hàng số cần quan tâm:

- chia cho factor (1, 10, 100, ...) để loại bỏ các hàng số bên phải của hàng số cần chú ý.

- lấy phần dư theo RADIX (10) kết quả phép chia trên để loại bỏ các hàng số bên trái của hàng số cần chú ý.

Như vậy cần $2n(\text{số hàng}) = O(n)$ tác vụ.

Trong mỗi bước, tất cả các số được chuyển vào các cột, sau đó gom lại đưa vào data, cũng cần $2n(\text{số hàng}) = O(n)$ phép chuyển.

Băm (Hashing)

Các phương thức tìm kiếm mô tả trong những chương trước dùng tác vụ chính là *so sánh các khóa*. Trong tìm kiếm tuần tự, mảng lưu trữ các phần tử liên tiếp nhau và khóa được so sánh để xác định phần tử cần tìm. Trong tìm kiếm nhị phân, khóa của phần tử giữa mảng được so sánh để xác định nửa mảng nào sẽ được kiểm tra tiếp tục hoặc tìm được phần tử cần tìm. Tương tự, trong cây BST, việc quyết định tiếp tục tìm kiếm trên một nhánh cụ thể được xác định bằng so sánh khóa.

Một cách tiếp cận khác cho tìm kiếm là tính toán vị trí của khóa trong bảng dựa trên *trị của khóa*. Khi biết khóa, vị trí của phần tử chứa khóa trong bảng có thể được tính toán từ trị khóa rồi *truy cập trực tiếp*, mà không cần phải qua hàng loạt kiểm tra như trong tìm kiếm nhị phân hoặc tìm kiếm trên cây BST. Điều này có nghĩa là thời gian tìm kiếm giảm từ $O(n)$ trong tìm kiếm tuần tự, hoặc từ $O(\lg n)$ trong tìm kiếm nhị phân, xuống còn $O(1)$. Bất kể kích thước của tập phần tử tìm kiếm, thời gian tìm kiếm luôn luôn bằng nhau. Trong trường hợp lý tưởng, trị của khóa biểu thị duy nhất vị trí của phần tử chứa khóa, trong thực tế ta chỉ cần xấp xỉ trường hợp lý tưởng là đủ tốt.

Ta cần tìm hàm $h()$ có thể chuyển đổi trị của khóa K (chuỗi, số, bản ghi, ...) thành một vị trí (chỉ số) trong bảng lưu trữ. Hàm $h()$ đó gọi là *hàm băm*. Nếu $h()$ chuyển các khóa khác nhau thành các địa chỉ khác nhau thì được gọi là *hàm băm hoàn hảo*.

Tuy nhiên, thực tế khó tạo được hàm băm hoàn hảo; với hai khóa khác nhau, hàm $h()$ vẫn có thể tạo một trị giống nhau, gọi là *đụng độ* (collision). Lúc này, hiệu quả của hàm băm phụ thuộc vào cách nó tránh đụng độ. Tránh đụng độ có thể đạt được bằng cách làm cho hàm $h()$ phức tạp hơn, nhưng sự phức tạp này dẫn đến chi phí tính toán $h(K)$ có thể rất cao, $h(K)$ ít phức tạp hơn thì có thể nhanh hơn.

I. Hàm băm (hash function)

Phân tích việc xây dựng hàm băm gán vị trí cho n khóa trong bảng có m vị trí ($n \leq m$). Mỗi khóa có thể gán m vị trí khác nhau, n khóa có khả năng m^n bố trí. Số hàm băm có thể là m^n . Để không có xung đột địa chỉ, khóa thứ nhất có m lựa chọn, khóa thứ hai có $m - 1$ lựa chọn, ... n khóa có $m!/(m - n)$ khả năng bố trí không đụng độ. Số lượng hàm băm hoàn hảo là $m!/(m - n)!$

Nếu lưu 50 khóa trong mảng có kích thước 100 phần tử, có $100^{50} = 10^{100}$ hàm băm trong đó chỉ có 10^{94} là hoàn hảo (một phần triệu). Phần lớn các hàm băm hoàn hảo này thực thi chậm và không thể biểu thị bằng một công thức ngắn gọn.

Vì vậy trong phần này chỉ thảo luận một số hàm băm hữu dụng, ít phức tạp hơn.

1. Phép chia (division)

Một hàm băm phải đảm bảo rằng trị vị trí mà nó trả về là một chỉ số hợp lệ chỉ đến một ô trong bảng. Cách đơn giản nhất là thực hiện phép chia lấy phần dư (modulo) cho TSize, $TSize = \text{sizeof(table)}$ là kích thước của bảng.

Khi đó $h(K) = K \bmod TSize$, nếu khóa K là một số. $TSize$ tốt nhất là số nguyên tố. Ngoài ra, $h(K) = (K \bmod p) \bmod TSize$ với số nguyên tố $p > TSize$ cũng có thể được dùng.

Phép chia thường được sử dụng cho hàm băm nếu biết ít thông tin về khóa.

2. Phép gấp (folding)

Khóa được chia thành vài phần. Các phần này được kết hợp lại, gọi là gấp với nhau, rồi được biến đổi một số cách để tạo ra địa chỉ lưu trữ đích. Có hai kiểu gấp: gấp dịch (shift folding) và gấp biên (boundary folding).

Khóa được chia thành vài phần và các phần này được xử lý bằng một tác vụ đơn giản như là cộng để kết hợp chúng lại.

- Trong shift folding, phần này được đặt dưới một phần khác và sau đó xử lý. Ví dụ, số SSN 123-45-6789 được chia thành ba phần là 123, 456, 789 và cộng các phần này lại. Kết quả là 1368, có thể chia lấy phần dư với TSize hoặc, nếu như kích thước của bảng là 1000 thì ba số đầu có thể được dùng như địa chỉ. Một khả năng khác là chia số 123-45-6789 thành năm phần (12, 34, 56, 78 và 9), cộng chúng lại rồi kết quả được chia lấy phần dư với TSize.

- Trong boundary folding, khóa được xem như là được viết trên một mẫu giấy rồi gấp lại với biên gấp nằm giữa các phần của khóa. Trong cách này, các phần khóa xen kẽ được đảo ngược thứ tự lại. Với ba phần của dãy số SSN: 123, 456, 789; phần thứ nhất, 123, được lấy theo đúng thứ tự, phần thứ hai được gấp ngược lại, 456 thành 654. Tiếp tục gấp ngược xuôi như vậy, phần thứ ba sẽ là 789. Kết quả sẽ là $123 + 654 + 789 = 1566$.

Trong cả hai phiên bản, khóa thường được chia thành những phần có kích thước cố định cộng với vài phần dư, và chúng được cộng với nhau. Quá trình xử lý đơn giản và nhanh, đặc biệt khi dùng các mẫu bit thay cho số nguyên. Nếu tính toán shift folding theo bit, dùng toán tử XOR (^) thay phép cộng.

Trong trường hợp khóa là chuỗi, một cách tiếp cận là XOR tất cả các ký tự của khóa và sử dụng kết quả làm địa chỉ. Ví dụ, đối với chuỗi "abcd", $h("abcd") = 'a' \wedge 'b' \wedge 'c' \wedge 'd'$. Tuy vậy, phương pháp đơn giản này cho kết quả là những địa chỉ giữa 0 và 127. Để có kết quả tốt hơn, những đoạn của chuỗi được XOR với đoạn khác, thay vì XOR từng ký tự.

3. Hàm Mid-Square

Trong phương pháp mid-square, khóa được tính trị bình phương (square) và trích lấy phần giữa (mid) của kết quả để làm địa chỉ. Nếu khóa là chuỗi, nó được xử lý trước để tạo ra một số, bằng cách nào đó, chẳng hạn dùng folding. Ví dụ, nếu khóa là 3121 thì $3121^2 = 9740641$, và đối với bảng có TSize bằng 1000, $h(3121) = 406$ là phần giữa của 3121^2 .

Thực tế, sẽ hiệu quả hơn khi chọn TSize là lũy thừa của 2, 2^n , và trích ra n bit từ phần giữa của dãy bit thể hiện bình phương của khóa. Giả sử kích thước của bảng là $1024 = 2^{10}$, thì trong ví dụ trên, trích 10 bit từ phần giữa dãy nhị phân của 3121^2 là phần in đậm giữa chuỗi bit 10010100**1010000101100001**. Phần giữa này, dãy nhị phân 0101000010, bằng 322. Phần này có thể được trích rút dễ dàng bằng cách sử dụng kết hợp mặt nạ bit và các toán tử trên bit.

4. Phép trích rút (extraction)

Trong phương pháp trích rút, chỉ một phần của khóa được dùng để tính địa chỉ. Đối với số SSN 123-45-6789, ta có thể dùng 2 số đầu của 4 số đầu, **1234**, và 2 số cuối của 4 số cuối, **6789**; kết hợp thành **1289**; hoặc vài cách kết hợp khác. Mỗi lần, chỉ có một phần của khóa được sử dụng. Nếu phần khóa này được chọn cẩn thận, nó đủ để băm, bỏ qua những phần để trùng lặp của khóa. Ví dụ, những số đầu của mã ISBN giống nhau với mọi cuốn sách được xuất bản bởi một nhà xuất bản. Chúng nên được loại bỏ khỏi việc tính địa chỉ nếu dữ liệu chỉ chứa sách của một nhà xuất bản.

5. Biến đổi cơ số (radix transformation)

Sử dụng biến đổi cơ số, khóa K được thể hiện trong một hệ thống số sử dụng cơ số khác. Nếu K là số thập phân 345_{10} , thì giá trị của nó trong hệ cơ số 9 (nonal) là 423_9 . Giá trị này sau đó được chia lấy phần dư với TSize, rồi được sử dụng làm địa chỉ. Tuy nhiên, không thể tránh khỏi đụng độ. Nếu TSize = 100, trị băm của 345 và 264 không đụng độ trong hệ cơ số 10, nhưng trong hệ cơ số 9 chúng lại đụng độ: $423_9 = 345_{10}$ và $323_9 = 264_{10}$, đều cho trị băm 23.

6. Hàm băm phổ dụng (universal hash function)

Khi biết quá ít về khóa, một lớp hàm băm gọi là hàm băm phổ dụng có thể được sử dụng. Một lớp của hàm băm được gọi là phổ dụng khi với bất kỳ mẫu nào, một thành viên được chọn ngẫu nhiên của lớp đó có phân phối mẫu như nhau, như vậy, thành viên của lớp đó đảm bảo xác suất đụng độ thấp.

Gọi H là một lớp các hàm ánh xạ từ một tập khóa tới một bảng băm có kích thước TSize. Ta nói rằng H là phổ dụng nếu với hai khóa bất kỳ x và y thuộc tập khóa, $x \neq y$, số các hàm băm h trong H thỏa điều kiện $h(x) = h(y)$ là $|H|/TSize$. Như vậy, H là phổ dụng nếu không có cặp khóa phân biệt nào được ánh xạ vào cùng một chỉ số (index) của bảng bởi một hàm h được chọn ngẫu nhiên với xác suất bằng $1/TSize$. Nói cách khác, có 1 khả năng trong TSize để 2 khóa đụng độ khi hàm băm ngẫu nhiên được áp dụng. Một lớp của hàm như vậy được định nghĩa như bên dưới.

Cho số nguyên tố $p \geq |\text{các khóa}|$, và chọn ngẫu nhiên hai số a và b,

$$H = \{h_{a,b}(K) : h_{a,b}(K) = ((aK + b) \bmod p) \bmod TSize \text{ và } 0 \leq a, b \leq p\}$$

Một ví dụ khác là một lớp khóa H dưới dạng các chuỗi byte, $K = K_0K_1\dots K_{r-1}$.

Ví dụ, số nguyên tố $p \geq 2^8 = 256$ và chuỗi $a = a_0, a_1, a_2, \dots, a_{r-1}$,

$$H = \{h_a(K) : h_a(K) = \left(\sum_{i=0}^{r-1} a_i K_i \right) \bmod p \bmod TSize \text{ và } 0 \leq a_0, a_1, a_2, \dots, a_{r-1} < p\}$$

II. Giải quyết đụng độ

Với hầu hết các hàm băm, vấn đề có nhiều hơn một khóa được đặt vào cùng một vị trí, gọi là đụng độ. Vấn đề này có thể được giải quyết bằng cách tìm một hàm băm cho trị băm phân phối đồng đều trong bảng hoặc tăng kích thước bảng. Hai yếu tố này, hàm băm và kích thước bảng, tuy có thể làm giảm đụng độ nhưng không thể hoàn toàn loại bỏ chúng.

Phần này thảo luận một số cách giải quyết đụng độ.

1. Đánh địa chỉ mở (open addressing)

Trong phương pháp đánh địa chỉ mở, khi một khóa đụng độ với một khóa khác thì đụng độ sẽ được giải quyết bằng cách tìm kiếm một vị trí còn trống trong bảng khác với vị trí gốc do khóa băm ra được. Nếu vị trí $h(K)$ đang bị chiếm, thì vị trí trong dãy thăm dò sau sẽ lần lượt được thử:

$$\text{norm}(h(K) + p(1)), \text{norm}(h(K) + p(2)), \dots, \text{norm}(h(K) + p(i)), \dots$$

Kết quả là tìm được chỗ trống, hoặc bảng đã đầy. Hàm p là 1 hàm thăm dò (probing function), i là biến thăm dò (probe), norm là hàm chuẩn hóa (normalization function) thường là phép chia lấy phần dư với kích thước của bảng (TSize).

a) Thăm dò tuyến tính (Linear Probing)

Phương pháp đơn giản nhất là thăm dò tuyến tính, trong đó $p(i) = i$, và với lần dò thứ i , vị trí được thử là $(h(K)+i) \bmod TSize$. Trong thăm dò tuyến tính, vị trí lưu trữ một khóa được xác định bằng cách tìm kiếm tuần tự tất cả các vị trí bắt đầu từ vị trí được tính toán bởi hàm băm đến khi tìm thấy một ô trống. Nếu tìm đến cuối bảng mà không thấy ô trống, quá trình tìm kiếm sẽ tiếp tục từ đầu bảng và dừng ngay trước $h(K)$, ô bắt đầu quá trình tìm kiếm.

A ₅ , A ₂ , A ₃	B ₅ , A ₉ , B ₂	B ₉ , C ₂	A ₅ , A ₂ , A ₃	B ₅ , A ₉ , B ₂	B ₉ , C ₂
0	0	0 B ₉	0	0	0 B ₉
1	1	1	1	1 B ₂	1 B ₂
2 A ₂	2 A ₂	2 A ₂	2 A ₂	2 A ₂	2 A ₂
3 A ₃	3 A ₃	3 A ₃	3 A ₃	3 A ₃	3 A ₃
4	4 B ₂	4 B ₂	4	4	4
5 A ₅	5 A ₅	5 A ₅	5 A ₅	5 A ₅	5 A ₅
6	6 B ₅	6 B ₅	6	6 B ₅	6 B ₅
7	7	7 C ₂	7	7	7
8	8	8	8	8	8 C ₂
9	9 A ₉	9 A ₉	9	9 A ₉	9 A ₉

Giải quyết đụng độ. Chỉ số dưới là mã băm của phần tử, dùng làm địa chỉ.

Trái: phương pháp thăm dò tuyến tính. Phải: phương pháp thăm dò bậc hai.

Thăm dò tuyến tính có xu hướng tạo thành cụm (cluster) trong bảng. Ví dụ, trong hình trên bên trái, khóa K_i được băm đến vị trí i . Đầu tiên, 3 khóa A_5, A_2 và A_3 được lưu đúng vị trí của chúng, gọi là vị trí nhà (home position). Sau đó, do khóa B_5 bị khóa A_5 chiếm vị trí home, B_5 được lưu vào vị trí tiếp theo đang trống. Tiếp theo, A_9 được lưu tại home, B_2 được lưu vào cách vị trí home của nó 2 đơn vị. Kế tiếp là B_9 , vị trí 9 không trống, và do nó là ô cuối trong bảng, quá trình thăm dò được bắt đầu từ đầu bảng, B_9 được lưu tại đây. Cuối cùng C_2 được lưu cách vị trí home của nó 5 đơn vị, hình thành cluster $A_2-A_3-B_2-A_5-B_5-C_2$.

Trong ví dụ này, xác suất lấp đầy những ô trống ngay sau cluster là $(\text{sizeof(cluster}) + 1)/TSize$, cao hơn những vị trí khác với xác suất được lấp đầy là $1/TSize$. Vì vậy, nếu cluster được tạo, nó có xu hướng lớn dần lên, cluster càng lớn thì khả năng tăng trưởng của nó càng cao. Điều này làm giảm hiệu năng lưu trữ và truy cập dữ liệu của bảng băm. Vấn đề là phải tránh việc hình thành của cluster. Giải pháp nằm trong việc lựa chọn cẩn thận hàm thăm dò p .

b) Thăm dò bậc hai

Một lựa chọn cho p là hàm bậc hai: $p(i) = h(K) + (-1)^{i-1}((i+1)/2)^2$ với $i = 1, 2, \dots, TSize - 1$

Công thức trên có thể chuyển thành chuỗi thăm dò đơn giản sau:

$$h(K) + i^2, h(K) - i^2 \text{ với } i = 1, 2, \dots, (TSize - 1)/2$$

Bao gồm luôn lần thử đầu tiên với $h(K)$, chuỗi thăm dò sẽ như sau:

$$h(K), h(K) + 1, h(K) - 1, h(K) + 4, h(K) - 4, \dots, h(K) + (TSize - 1)^2/4, h(K) - (TSize - 1)^2/4$$

tất cả đều chia lấp phần dư với $TSize$. Kích thước bảng $TSize$ không nên là số chẵn, vì chỉ có vị trí chẵn hoặc vị trí lẻ được thử tùy thuộc vào giá trị của $h(K)$. Kích thước bảng lý tưởng nên là số nguyên tố $4j + 3$ của số nguyên j , điều này bảo đảm tất cả các vị trí đều có trong chuỗi thăm dò.

Chú ý rằng công thức xác định chuỗi thăm dò không phải là $h(K) + i^2$, với $i = 1, 2, \dots, TSize - 1$. Nếu như vậy, chuỗi thăm dò chỉ quan tâm một nửa của bảng, chúng duyệt đi rồi duyệt lại nửa bảng đó theo thứ tự ngược lại.

Ví dụ với $TSize = 19$, $h(K) = 9$, chuỗi thăm dò: 9, 10, 13, 18, 6, 15, 7, 1, 16, 14, 14, 16, 1, 7, 15, 6, 18, 13, 10

Mặc dù sử dụng thăm dò bậc hai cho kết quả tốt hơn so với thăm dò tuyến tính, vẫn đề kết cụm (cluster) không tránh được hoàn toàn, bởi vì với các khóa băm vào cùng vị trí, chuỗi thăm dò chúng sử dụng là giống nhau. Cluster trong phương pháp này được gọi là cluster thứ cấp (secondary cluster). Tuy nhiên, chúng ít tác hại hơn các cluster sơ cấp.

c) Thăm dò ngẫu nhiên

Một khả năng khác, p là bộ sinh số ngẫu nhiên (Morris, 1968). Phương pháp này không tạo cluster, nhưng để tìm lại khóa, bộ sinh số ngẫu nhiên phải được khởi tạo với cùng "mầm ngẫu nhiên" (seed) để cho cùng khóa trước khi sinh ra chuỗi thăm dò.

d) Băm kép (double hashing)

Vấn đề cluster thứ cấp được giải quyết tốt với băm kép. Phương pháp này sử dụng hai hàm băm: $h()$ dùng truy cập vào vị trí home của một khóa; và $h_p()$ dùng giải quyết đụng độ. Chuỗi thăm dò trở thành:

$$h(K), h(K) + h_p(K), \dots, h(K) + ih_p(K), \dots$$

tất cả đều chia lấp phần dư với $TSize$. Kích thước bảng $TSize$ nên là số nguyên tố để mỗi vị trí trong bảng đều có trong chuỗi thăm dò. Thực nghiệm cho thấy cluster thứ cấp được loại bỏ do chuỗi thăm dò phụ thuộc vào h_p , mà h_p lại phụ thuộc vào khóa.

Tuy nhiên, cần lựa chọn h_p cẩn thận, ví dụ $h_p(K) = \text{length}(K)$ có thể sinh chuỗi thăm dò giống nhau.

Ngoài ra, sử dụng băm kép có thể tốn thêm thời gian, nếu hàm băm được dùng phức tạp. Để đơn giản, hàm băm thứ hai có thể định nghĩa dựa trên hàm băm thứ nhất, chẳng hạn: $h_p(K) = ih(K) + 1$.

Các phương pháp đã thảo luận đều dựa trên kích thước của bảng và số lượng phần tử đã đặt vào bảng. Tính không hiệu quả của các phương pháp trên thể hiện ở số thao tác tìm kiếm không thành công. Có càng nhiều phần tử trong mảng thì cluster (sơ cấp hay thứ cấp) dễ hình thành và tăng kích thước.

Xét trường hợp thăm dò tuyến tính. Nếu K không có trong bảng, bắt đầu từ vị trí $h(K)$, tất cả các ô không rỗng theo sau sẽ được kiểm tra; cluster càng dài thì thời gian để xác định việc K không có trong bảng càng mất nhiều thời gian. Trong trường hợp xấu, tức là khi bảng đầy, ta phải kiểm tra tất cả các ô kể từ $h(K)$ cho đến $(h(K) - 1) \bmod TSize$. Như vậy, thời gian tìm kiếm tăng tỷ lệ thuận với số phần tử đã có trong bảng.

Donald Knuth (1998) phát triển các công thức dùng tính xấp xỉ số bước tìm kiếm thành công hoặc số bước tìm kiếm thất bại cho các phương pháp băm khác nhau.

	Thăm dò tuyến tính	Thăm dò bậc hai	Băm kép
Tìm kiếm thành công	$\frac{1}{2} \left(1 + \frac{1}{1-LF} \right)$	$1 - \ln(1-LF) - \frac{1}{2}$	$\frac{1}{LF} \ln \frac{1}{1-LF}$
Tìm kiếm không thành công	$\frac{4}{2} \left(1 + \frac{1}{(1-LF)^2} \right)$	$\frac{1}{1-LF} - LF - \ln(1-LF)$	$\frac{1}{1-LF}$

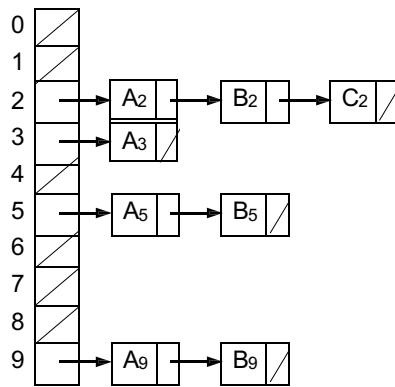
LF (Load Factor) gọi là hệ số tải của bảng, tính bằng công thức: $LF = \text{số phần tử đang có trong bảng} / \text{kích thước của bảng}$. LF càng bé, các công thức tính xấp xỉ trên càng gần với thực tế. Từ công thức trên, thăm dò tuyến tính cần 35% bảng trống để duy trì hoạt động ở mức độ chấp nhận được. Tỷ lệ phần trăm này sẽ thấp hơn đối với thăm dò bậc hai (25%) và băm kép (20%), nhưng vẫn bị xem là lãng phí, nhất là khi bảng lưu trữ quá lớn.

Nếu bảng quá đầy, các tác vụ sẽ thực hiện chậm và tác vụ chèn có thể thất bại, nhất là với thăm dò bậc hai. Một giải pháp là xây dựng bảng mới với kích thước ($TSize$) gấp đôi bảng cũ, với một hàm băm mới liên quan (do $TSize$ thay đổi). Sau đó quét toàn bộ bảng băm gốc, tính toán trị băm cho các phần tử (không bị xóa) và chèn nó vào bảng băm mới. Giải pháp này gọi là băm lại (rehashing).

2. Tạo dây chuyền (chaining)

Các khóa không nhất thiết phải lưu trong bảng. Trong phương pháp tạo dây chuyền, mỗi vị trí của bảng chứa một tham chiếu đến một danh sách liên kết lưu trữ các khóa đụng độ. Phương pháp này được gọi là tạo dây chuyền tách biệt (separate chaining), và bảng chứa các tham chiếu được gọi là *bảng scatter* (bảng rải). Trong phương pháp này, bảng không bao giờ bị tràn, vì danh sách liên kết dễ dàng mở rộng khi có khóa đụng độ mới đi vào. Tuy nhiên, việc tăng độ dài của những danh sách liên kết này có thể ảnh hưởng đến hiệu suất tìm kiếm. Hiệu suất tìm kiếm có thể được cải tiến bằng cách duy trì một thứ tự sắp xếp cho tất cả các danh sách liên kết hoặc sử dụng danh sách liên kết tự tổ chức (self-organizing).

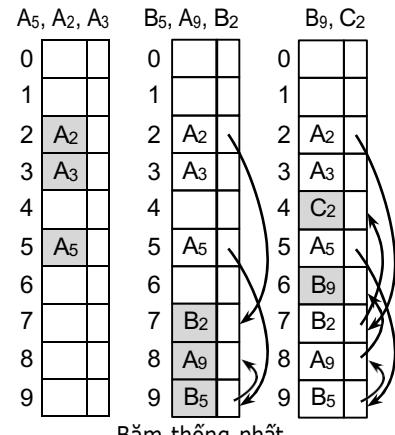
Phương pháp này cần không gian bổ sung cho việc lưu các tham chiếu. Bảng scatter chỉ chứa các tham chiếu, còn mỗi node ngoài khóa được lưu cần có một trường tham chiếu chỉ đến node kế tiếp. Vì thế, với n khóa, cần $n + TSize$ tham chiếu.



Phương pháp tạo dây chuyền dùng danh sách liên kết

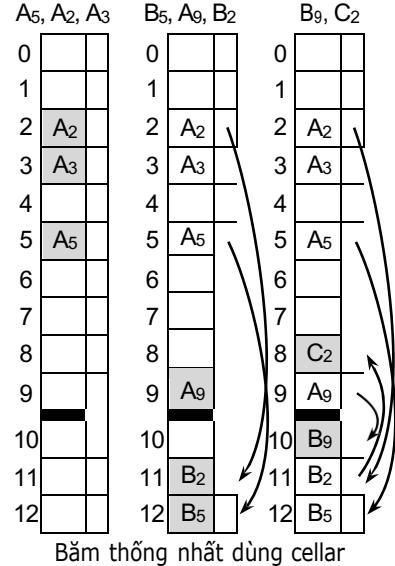
Chèn các phần tử A5, A2, A3, B5, A9, B2, B9, C2

Một phiên bản khác của phương pháp tạo dây chuyền được gọi là hàm băm thống nhất (coalesced hashing) hoặc tạo dây chuyền thống nhất, coalesced chaining) kết hợp thăm dò tuyến tính với tạo dây chuyền. Trong phương pháp này, vị trí trống đầu tiên kể từ cuối bảng được dùng để lưu khóa đụng độ với khóa khác, và chỉ số của vị trí này được lưu cùng với khóa đụng độ đã có mặt trước đó trong bảng; kết quả giống như một danh sách liên kết. Bằng cách này, có thể tránh việc tìm kiếm tuyến tính đi xuống bảng cách tìm kiếm theo danh sách liên kết. Mỗi vị trí pos của bảng bao gồm hai trường: info lưu trữ khóa và next lưu trữ chỉ số của khóa đụng độ tiếp theo (khóa cho cùng kết quả băm là pos). Trường next này có thể đánh dấu -1 để chỉ ra điểm kết thúc của dây chuyền. Phương pháp này cần TSize(sizeof(reference) + sizeof(next)) không gian bổ sung ngoài không gian dành cho các khóa. Không gian bổ sung này ít hơn so với phương pháp tạo dây chuyền, nhưng kích thước bảng lại giới hạn số khóa có thể băm vào bảng.



Băm thống nhất

Một vùng tràn (overflow area), gọi là hầm (cellar) có thể được cấp phát thêm để lưu trữ các khóa đụng độ.



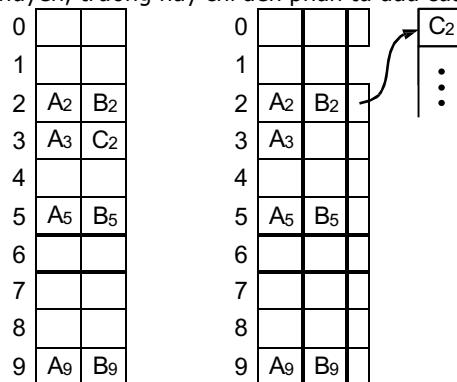
Băm thống nhất dùng cellar

3. Đánh địa chỉ xô (bucket addressing)

Một giải pháp khác cho vấn đề đụng độ là chứa các khóa đụng độ trong cùng vị trí trên bảng. Điều này có thể đạt được bằng cách kết hợp một xô (bucket) với mỗi địa chỉ. Một xô là một khối không gian đủ lớn để có thể chứa nhiều phần tử.

Dùng phương pháp đánh địa chỉ xô vẫn chưa tránh khỏi hoàn toàn vấn đề đụng độ. Nếu một xô đã đầy, một khóa được băm tới nó phải được chứa ở một nơi nào khác. Có thể kết hợp với phương pháp đánh địa chỉ mở, sử dụng thăm dò tuyến tính, lưu khóa đụng độ trong xô tiếp theo nếu xô này còn chỗ trống. Khóa đụng độ cũng có thể được lưu trong một xô khác, nếu phương pháp thăm dò bậc hai được sử dụng.

Những khóa đụng độ có thể được lưu trong vùng tràn (overflow area). Trong trường hợp đó, mỗi xô có thêm một trường chỉ tới vùng tràn. Nếu kết hợp với phương pháp chèn, trường này chỉ đến phần tử đầu của danh sách liên kết nối với xô.



Phương pháp đánh địa chỉ xô. Chèn các phần tử A₅, A₂, A₃, B₅, A₉, B₂, B₉, C₂
Trái: kết hợp thăm dò tuyến tính. Phải: dùng vùng tràn.

III. Loại bỏ phần tử

Với phương pháp tạo dây chuyền, việc xóa một phần tử sẽ dẫn đến việc xóa một node trong danh sách liên kết chứa phần tử đó. Với các phương pháp khác, tác vụ loại bỏ phần tử có thể phải xử lý thận trọng hơn.

Xem ví dụ, các khóa được lưu trữ bằng phương pháp thăm dò tuyến tính, theo thứ tự: A₁, A₄, A₂, B₄, B₁. Sau khi A₄ bị xóa và vị trí thứ 4 trở nên trống, ta thử tìm B₄ bằng việc kiểm tra vị trí thứ 4 trước tiên. Do vị trí này đang trống nên chúng ta kết luận sai rằng B₄ không có trong bảng. Kết quả tương tự khi ta xóa A₂, thao tác tìm kiếm B₁ sẽ không thành công, bởi vì thao tác thăm dò tuyến tính sẽ dừng ngay vị trí thứ 2. Tình trạng tương tự xảy ra cho các phương pháp đánh địa chỉ mở (open addressing) khác. Ta có thể để lại các khóa bị xóa và đánh dấu không hợp lệ cho chúng, gọi là đặt tombstone, việc tìm kiếm sẽ không bị dừng khi gặp tombstone, các phần tử tiếp sau nó vẫn được tìm thấy. Khi một khóa mới một chèn vào, nó được quyền ghi đè lên tombstone. Tuy nhiên, đối với trường hợp số lượng xóa lớn và số lượng chèn thêm vào ít, bảng sẽ bị quá tải với các tombstone, dẫn đến tăng thời gian tìm kiếm. Vì thế, bảng nên được lọc sau một số lượng thao tác xóa nhất định bằng cách dồn các phần tử còn lại trong chuỗi thăm dò để lấp các tombstone.

A ₁ , A ₄ , A ₂ , B ₄ , B ₁	Xóa A ₄	Xóa A ₂	Lọc bảng
0	0	0	0
1 A ₁	1 A ₁	1 A ₁	1 A ₁
2 A ₂	2 A ₂	2 RIP	2 B ₁
3 B ₁	3 B ₁	3 B ₁	3
4 A ₄	4 RIP	4 RIP	4 B ₄
5 B ₄	5 B ₄	5 B ₄	5
6	6	6	6
7	7	7	7
8	8	8	8
9	9	9	9

Xóa các phần tử trong bảng trong thăm dò tuyến tính dùng tombstone (RIP).

IV. Hàm băm hoàn hảo

Hàm băm hiếm khi là hàm băm lý tưởng, theo nghĩa nó có thể băm một khóa thành một địa chỉ riêng biệt, tránh mọi đụng độ. Trong đa số trường hợp, kỹ thuật giải quyết đụng độ phải được thêm vào, bởi vì không sớm thì muộn, một khóa đưa vào sẽ đụng độ với khóa đang có trong bảng. Kích thước của bảng cũng ảnh hưởng đến khả năng đụng độ, để tránh đụng độ nó phải đủ lớn do khó biết trước được số lượng khóa.

Nguyên nhân gây ra vấn đề trên là do ta không biết nội dung của dữ liệu trước khi băm chúng. Thông thường, hàm băm được xây dựng trước rồi sau đó dữ liệu mới được băm. Tuy vậy, trong nhiều tình huống, ta biết trước nội dung của dữ liệu cần băm, hàm băm được xây dựng trên dữ liệu đã biết nội dung nên có thể hoàn hảo.

Một hàm băm được xem là hoàn hảo (perfect hash function) nếu nó băm ra ngay địa chỉ lưu trữ trong lần thử đầu tiên. Thêm vào đó, nếu một hàm băm chỉ cần số ô trong bảng bằng với số khóa, sao cho khi băm các khóa xong không còn ô nào còn trống thì hàm băm đó được gọi là hàm băm hoàn hảo tối thiểu (minimal perfect hash function). Dùng hàm băm hoàn hảo tối thiểu, ta tránh được tiêu tốn thời gian cho giải quyết đụng độ và lãng phí do những ô trống không dùng đến trong bảng.

1. Phương pháp Cichelli

Một giải thuật dùng để tìm ra hàm băm hoàn hảo tối thiểu được phát triển bởi Richard J. Cichelli. Nó được dùng để băm một số nhỏ các khóa, ví dụ như các từ khóa (keyword) dành riêng cho một ngôn ngữ lập trình.

Hàm được viết dưới dạng: $h(\text{word}) = [\text{length}(\text{word}) + g(\text{firstletter}(\text{word})) + g(\text{lastletter}(\text{word}))] \bmod \text{TSize}$

trong đó hàm $g()$ được xây dựng bằng cách dùng giải thuật Cichelli. Hàm $g()$ gán trị (gọi là g-value) cho các ký tự sao cho hàm kết quả $h()$ sẽ trả về một trị băm duy nhất cho mỗi khóa trong tập hợp khóa cần băm. Trị mà hàm $g()$ gán cho một ký tự cụ thể không nhất thiết phải duy nhất.

Giải thuật có 3 giai đoạn:

- Tính toán số lần xuất hiện ký tự đầu và cuối của các khóa.

- Sắp xếp các khóa theo số lần xuất hiện của ký tự đầu và cuối.
- Tìm kiếm, bước cuối cùng và là trọng tâm của giải thuật: thủ tục search() đệ quy có dùng hàm hỗ trợ try().

cichelliMethod()

chọn một trị cho max;

(1) tính số lần xuất hiện của mỗi ký tự đầu (first) và ký tự cuối (last) trong tập tất cả các khóa;

(2) sắp xếp tất cả các khóa theo tổng số lần xuất hiện của first và last cho mỗi khóa;

(3) search (wordList)

```

if wordList rỗng
    dừng xử lý;
word = từ đầu tiên của wordList;
wordList = wordList với từ đầu tiên để riêng ra;
if first và last của word đã được gán g-value
    try(word, -1, -1);
    if success
        search (wordList);
    đặt word tại đầu wordList và để riêng trị hash của nó ra;
else if first hoặc last của word không có g-value
    for mỗi n, m trong {0, ... , max}
        try(word, n, m);
        if success
            search (wordList);
        đặt word tại đầu wordList và để riêng trị hash của nó ra;
else if first hoặc last của word có g-value
    for mỗi n trong {0, ... , max}
        try(word, -1, n) hoặc try(word, n, -1);
        if success
            search (wordList);
        đặt word tại đầu wordList và để riêng trị hash của nó ra;

```

try(word, firstValue, lastValue)

```

tính h(word);
if h(word) không hợp lệ // đúng độ
    để trống h(word);
    gán firstValue và/hoặc lastValue như là g-value của first(word) và/hoặc last(word)
if g-value của first hoặc last khác -1 // không bị gán bởi lệnh trên
    return success;
    return failure;

```

Ví dụ: băm các khóa trong danh sách: calliope, clio, erato, euterpe, melpomene, polyhymnia, terpsichore, thalia, urania

			trị băm dành trước
euterpe	e = 0	h = 7	{7}
calliope	c = 0	h = 8	{7 8}
erato	o = 0	h = 5	{5 7 8 }
terpsichore	t = 0	h = 2	{2 5 7 8}
melpomene	m = 0	h = 0	{0 2 5 7 8}
thalia	a = 0	h = 6	{0 2 5 6 7 8}
clio		h = 4	{0 2 4 5 6 7 8}
polyhymnia	p = 0	h = 1	{0 1 2 4 5 6 7 8}
urania	u = 0	h = 6 *	{0 1 2 4 5 6 7 8}
urania	u = 1	h = 7 *	{0 1 2 4 5 6 7 8}
urania	u = 2	h = 8 *	{0 1 2 4 5 6 7 8}
urania	u = 3	h = 0 *	{0 1 2 4 5 6 7 8}
urania	u = 4	h = 1 *	{0 1 2 4 5 6 7 8}
polyhymnia	p = 1	h = 2 *	{0 2 4 5 6 7 8}
polyhymnia	p = 2	h = 3	{0 2 3 4 5 6 7 8}
urania	u = 0	h = 6 *	{0 2 3 4 5 6 7 8}
urania	u = 1	h = 7 *	{0 2 3 4 5 6 7 8}
urania	u = 2	h = 8 *	{0 2 3 4 5 6 7 8}
urania	u = 3	h = 0 *	{0 2 3 4 5 6 7 8}
urania	u = 4	h = 1	{0 1 2 3 4 5 6 7 8}

Giai đoạn 3 của giải thuật Cichelli, chuỗi các lời gọi thủ tục search() với max = 4

g-value được gán cho các ký tự, dấu * là các trị băm bị từ chối

- (1) Xác định số lần xuất hiện của các ký tự đầu và cuối của các khóa: e(6), a(3), c(2), o(2), t(2), m(1), p(1), u(1)
- (2) Sắp xếp các khóa:

Tính điểm cho khóa bằng cách cộng số lần xuất hiện ký tự đầu và cuối của mỗi khóa.

calliope(8), clio(4), erato(8), euterpe(12), melpomene(7), polyhymnia(4), terpsichore(8), thalia(5), urania(4)

Sau đó sắp xếp các khóa theo điểm, ta nhận được wordList dùng cho giai đoạn 3.

euterpe, calliope, erato, terpsichore, melpomene, thalia, clio, polyhymnia, urania

- (3) Tìm kiếm: thủ tục search()

Thủ tục search được áp dụng với max = 4. Đầu tiên khóa euterpe được thử, ký tự đầu và cuối "e" được gán g-value là 0, tính được $h(euterpe) = 7$, 7 được đặt vào danh sách các giá trị băm dành trước. Các khóa khác được xử lý tương tự cho đến lượt khóa urania. Tất cả năm g-value có thể (0 - 4) gán cho "u" đều cho $h(urania)$ đã có trong danh sách các trị băm. Thủ tục đệ quy giúp quay lại (backtracking) bước trước đó, khi polyhymnia được thử. Trị băm của polyhymnia được loại khỏi danh sách các trị băm và g-value là 1 được gán cho "p", dẫn đến lỗi; nhưng g-value là 2 gán cho "p" lại cho trị băm $h(polyhymnia) = 3$ hợp lệ nên giải thuật tiếp tục. Khóa urania được thử lại, đến lần thứ 5 thì thành công với $h(urania) = 1$. Các từ khóa đều được gán trị băm duy nhất, thủ tục search kết thúc. Như vậy, với g-value:

$$a = c = e = o = m = t = 0, p = 2 \text{ và } u = 4$$

hàm $h()$ sẽ là hàm băm hoàn hảo tối thiểu trên tập 9 từ khóa trên.

Tiến trình tìm kiếm trong giải thuật Cichelli là lũy thừa bởi vì nó dùng tìm kiếm vét cạn (exhaustive search); vì vậy, không thể áp dụng giải thuật cho những số lượng khóa lớn. Tuy nhiên, đối với số lượng khóa nhỏ, nó lại cho kết quả khá tốt.

2. Giải thuật FHCD

Giải thuật FHCD (Edward I. Fox, Lenwood S. Heath, Qi Fan Chen, Amjad M. Daoud) là một biến thể của phương pháp Thomas Sager, phát triển từ phương pháp Cichelli. Giải thuật FHCD tìm hàm băm hoàn hảo tối thiểu theo công thức:

$$h(\text{word}) = [h_0(\text{word}) + g(h_1(\text{word})) + g(h_2(\text{word}))] \bmod T\text{Size}$$

$g()$ là hàm được xác định bởi giải thuật. Để định nghĩa hàm h_i , ba bảng chứa số ngẫu nhiên được định nghĩa; T_0 , T_1 và T_2 cho mỗi hàm h_i . Mỗi word là một chuỗi ký tự $c_1c_2 \dots c_m$ tương ứng với bộ ba (h_0, h_1, h_2) tính bởi công thức sau:

$$h_0 = (T_0(c_1) + \dots + T_0(c_m)) \bmod n$$

$$h_1 = (T_1(c_1) + \dots + T_1(c_m)) \bmod r$$

$$h_2 = ((T_2(c_1) + \dots + T_2(c_m)) \bmod r) + r$$

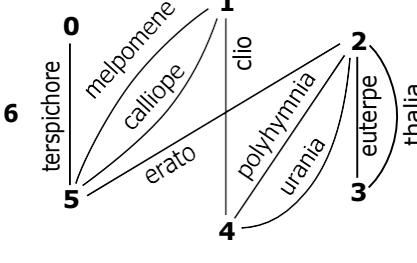
n là số các word có trong dữ liệu, r là tham số, thường bằng hoặc nhỏ hơn $n/2$, và $T_i(c_j)$ là số ngẫu nhiên trong bảng T_i cho c_j .

Ví dụ minh họa, giống như trong phần phương pháp Cichelli, băm các khóa trong danh sách:

calliope, clio, erato, euterpe, melpomene, polyhymnia, terpsichore, thalia, urania

Tri của:	h_0	h_1	h_2
calliope	(0	1	5)
clio	(7	1	4)
erato	(3	2	5)
euterpe	(6	2	3)
melpomene	(3	1	5)
polyhymnia	(8	2	4)
terpsichore	(8	0	5)
thalia	(8	2	3)
urania	(0	2	4)

(a) Level Vertex g-value



(b)

Level	Vertex	Cung
0	$v_1[2]$	
1	$v_2[5]$	erato
2	$v_3[1]$	calliope, melpomene
3	$v_4[4]$	clio, polyhymnia, urania
4	$v_5[3]$	euterpe, thalia
5	$v_6[0]$	terpsichore

(c) Level Vertex g-value

x	$g(x)$
0	4
1	4
2	2
3	4
4	2
5	6

(e)

Các bước của giải thuật FHCD

Hàm $g()$ được xác định trong ba bước của giải thuật:

- Giai đoạn ánh xạ (mapping): n bộ ba $(h_0(\text{word}), h_1(\text{word}), h_2(\text{word}))$ khác nhau được tạo. Sau đó, một đồ thị hai phía (bipartite graph) gọi là *đồ thị phụ thuộc* (dependency graph) được xây dựng. Phân nửa các đỉnh của nó tương ứng với các trị h_1 (được đặt nhãn 0 đến $r - 1$), phân nửa còn lại tương ứng các trị h_2 (được đặt nhãn r đến $2r - 1$). Như vậy, một khóa word sẽ tương ứng với một cạnh của đồ thị kết nối đỉnh $h_1(\text{word})$ và đỉnh $h_2(\text{word})$.

Hình (a) trình bày 9 bộ ba ngẫu nhiên, tương ứng với 9 khóa. Hình (b) là đồ thị phụ thuộc tương ứng với $r = 3$, ví dụ khóa erato có $h_1(erato) = 2$ và $h_2(erato) = 5$, erato tương ứng với cạnh 2 - 5.

- Giai đoạn sắp xếp (ordering): sắp xếp các đỉnh theo thứ tự bậc của đỉnh giảm dần, thành chuỗi các đỉnh v_1, \dots, v_t . Phân phôi khóa vào các level, ta gọi level $K(v_i)$ là tập hợp các khóa tương ứng các cạnh kết nối đỉnh v_i với các đỉnh v_j , sao cho $j \leq i$.

Hình (c), phân phôi khóa vào các level. Ví dụ: level $K(v_4)$, chứa các cạnh kết nối đỉnh v_4 (tức đỉnh chứa số 4) với các đỉnh trước nó trong chuỗi các đỉnh: $v_4[4]-v_4[4]$ (không có), $v_4[4]-v_3[1]$ (clio), $v_4[4] - v_2[5]$ (không có), $v_4[4] - v_1[2]$ (polyhymnia, urania).

- Giai đoạn tìm kiếm (searching): xem hình (d) và (e). Đầu tiên chọn $g(v_1)$ ngẫu nhiên bằng 2, nghĩa là $g(v_1) = g(2) = 2$. Đỉnh tiếp $v_2 = 5$, $K(v_2) = \{\text{erato}\}$, ta tính trị băm của erato theo công thức:

$$h(\text{erato}) = [h_0(\text{erato}) + g(h_1(\text{erato})) + g(h_2(\text{erato}))] \bmod T\text{Size} = [3 + g(2) + g(5)] \bmod 9$$

$g(2) = 2$, $g(5)$ lấy ngẫu nhiên bằng 6, $h(\text{erato}) = 2$. Tiếp theo $v_3 = 1$, $K(v_3) = \{\text{calliope}, \text{melpomene}\}$.

$$h(\text{calliope}) = [0 + g(1) + 6] \bmod 9 \text{ và } h(\text{calliope}) = [3 + g(1) + 6] \bmod 9$$

ta thử tìm $g(1)$ ngẫu nhiên sao cho $h(\text{calliope})$ và $h(\text{melpomene})$ khác 2, đã bị chiếm trong bảng, ta chọn $g(1) = 4$.

Tiếp tục tương tự với các khóa khác.

Nén dữ liệu

Truyền thông tin là chức năng cơ bản của nhiều hệ thống. Truyền thông tin luôn yêu cầu tốc độ truyền nhanh hơn. Ta có thể cải thiện tốc độ truyền bằng cách cải thiện kênh truyền hoặc cải thiện dữ liệu truyền sao cho nó vẫn chứa cùng thông tin nhưng có kích thước dữ liệu truyền nhỏ hơn.

I. Điều kiện nén dữ liệu

Nén dữ liệu (data compression, hoặc compaction) sẽ thu giảm kích thước dữ liệu truyền mà không ảnh hưởng đến thông tin chứa trong nó.

Các ký tự m_i dùng trong ngôn ngữ M gọi là các symbol của tập M. Giả sử m_i được lựa chọn một cách độc lập, có xác suất xuất hiện $P(m_i)$, các symbol được mã hóa thành codeword chứa 0 và 1.

Độ bất định (entropy) của nguồn M được định nghĩa:

$$L_{ave} = P(m_1)L(m_1) + \dots + P(m_n)L(m_n)$$

ở đây $L(m_i) = -\lg(P(m_i))$ là kích thước tối thiểu của codeword cho symbol m_i . Công thức này cho biết độ dài trung bình tốt nhất có thể có của một codeword khi các symbol và xác suất của chúng được biết. Nói cách khác, L_{ave} dùng để đánh giá chất lượng codeword dùng để nén, ta tính độ dài trung bình của codeword và so sánh với L_{ave} , càng gần với trị này, tỷ lệ nén càng tốt.

Vài kỹ thuật nén dữ liệu thử tối thiểu hóa L_{ave} bằng cách tạo mã tối ưu. Một số nguyên tắc khi xây dựng mã tối ưu:

- Một codeword chỉ tương ứng với một symbol.
- Thuộc tính tiền tố (prefix property, còn gọi là prefix-free): không có codeword nào là tiền tố (prefix) của một codeword khác.

Xem ba symbol trong bảng sau:

Symbol	Code ₁	Code ₂	Code ₃
A	1	1	11
B	2	22	12
C	12	12	21

Code₁ không cho phép phân biệt giữa AB và C, cả hai đều được mã như 12. Code₂ không gây nhầm lẫn như Code₁, nhưng nó cần thao tác dò trước (look ahead). Ví dụ 1222, ta chọn AB và phát hiện dư mã 2; nên chọn CB mới đúng. Cả hai loại code này vi phạm thuộc tính tiền tố. Code₃ không vi phạm thuộc tính tiền tố.

- Symbol xuất hiện ít có codeword dài và ngược lại. Nghĩa là, nếu $P(m_i) \leq P(m_j)$, thì $L(m_i) \geq L(m_j)$, với $1 \leq i, j \leq n$.
- Tân dụng các codeword có kích thước ngắn chưa dùng đến mà không vi phạm thuộc tính tiền tố, trước khi dùng đến các codeword có kích thước dài. Ví dụ, tập 5 symbol mã hóa thành các codeword sau: 01, 000, 001, 100, 101 là không tối ưu vì chưa dùng đến mã 11. Có thể mã hóa với codeword tối ưu hơn: 01, 10, 11, 000, 001.

Khi so sánh hiệu quả của các phép nén khác nhau trên cùng dữ liệu nén, ta dùng tỷ lệ nén:

$$\frac{\text{length}(pu)}{\text{length}(tpu)}$$

Tỷ lệ nén này tính bằng % cho thấy độ dư thừa được thu giảm từ dữ liệu đầu vào.

II. Mã hóa Huffman

1. Giải thuật Huffman

Một loại mã tối ưu, theo các nguyên tắc trên, được phát triển bởi David Huffman, dùng cây nhị phân và cho codeword nhị phân. Giải thuật như sau:

Huffman()

for mỗi symbol

 tạo một cây (có node gốc chứa xác suất xuất hiện của symbol)

 và sắp các cây với thứ tự tăng theo xác suất xuất hiện của symbol;

 while còn hơn một cây

 lấy hai cây t_1, t_2 với xác suất xuất hiện nhỏ nhất p_1, p_2 ($p_1 \leq p_2$);

 tạo cây với t_1 và t_2 là con của nó, xác suất chứa tại node gốc mới bằng $p_1 + p_2$;

 liên kết 0 với từng nhánh trái và 1 với từng nhánh phải;

 tạo một codeword duy nhất cho mỗi symbol bằng cách duyệt cây từ node gốc đến

 node lá chứa xác suất tương ứng với symbol đó và tạo chuỗi từ các số 0 và 1 gấp được;

Cây kết quả có xác suất tại node gốc là 1. Chú ý rằng do xác suất của các cây có thể bằng nhau, giải thuật có nhiều cách lựa chọn hai cây có xác suất nhỏ nhất nên có thể tạo ra nhiều cây có hình dáng khác nhau. Tuy nhiên, dù hình dáng cây khác nhau, codeword sinh ra khác nhau, chiều dài trung bình của codeword do chúng tạo ra vẫn như nhau.

Để đo lường hiệu quả của giải thuật Huffman, dùng định nghĩa chiều dài đường đi có trọng số (weighted path length). Định nghĩa này giống định nghĩa entropy, ngoại trừ $L(m_i)$ là kích thước codeword được gán cho symbol m_i bởi giải thuật này.

Trong ví dụ minh họa bên dưới, cây Huffman được tạo cho ngôn ngữ có 5 ký tự A, B, C, D và với xác suất tương ứng là .39, .21, .19, .12 và .09. Có hai cây được tạo ra, chúng có codeword khác nhau nhưng chiều dài của codeword gán cho các symbol tương ứng là như nhau.

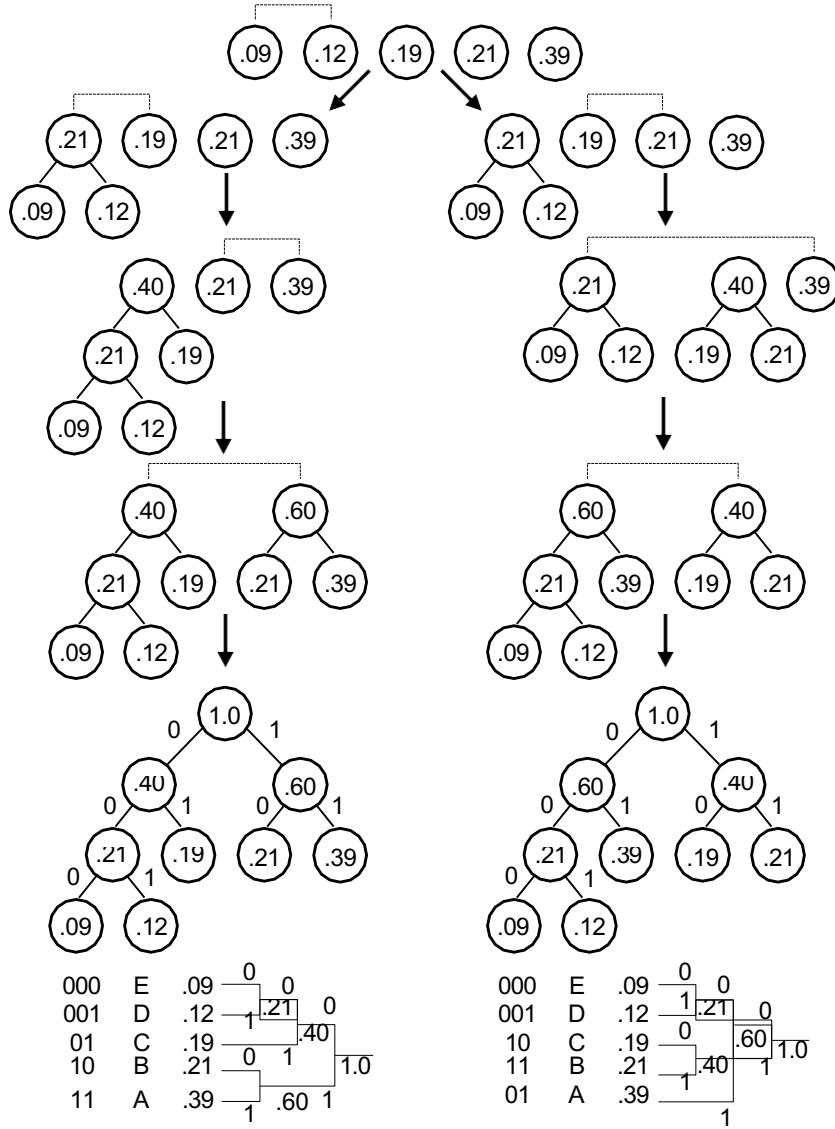
Chiều dài trung bình của codeword, tính theo định nghĩa chiều dài đường đi có trọng số là:

$$L_{Huf} = (.39 \times 2) + (.21 \times 2) + (.19 \times 2) + (.12 \times 3) + (.09 \times 3) = 2.21$$

Trong lúc chiều dài trung bình tính theo công thức entropy là:

$$L_{ave} = .39 \times 1.238 + .21 \times 2.252 + .19 \times 2.396 + .12 \times 3.059 + .09 \times 3.474 = 2.09$$

Ta có $\text{diff}(L_{Huf}, L_{ave}) = (2.21, 2.09) = 5\%$, cho thấy kết quả nén rất tốt.

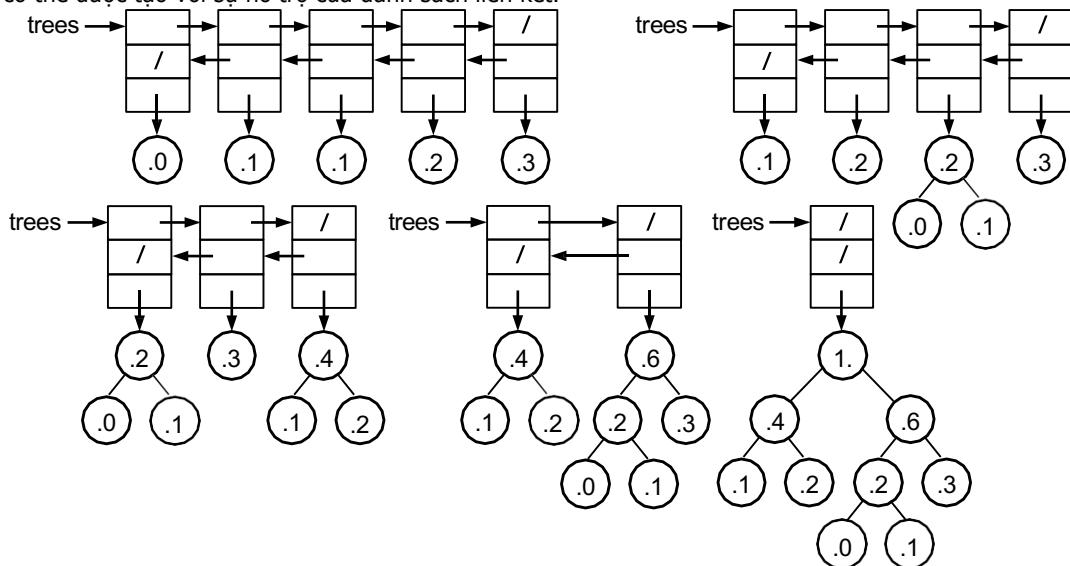


Hai cây Huffman được tạo ra từ 5 ký tự A, B, C, D và E, với xác suất tương ứng .39, .21, .19, .12 và .09

2. Tạo cây Huffman

a) Dùng danh sách liên kết

Cây Huffman có thể được tạo với sự hỗ trợ của danh sách liên kết.



Tạo cây Huffman, dùng danh sách liên kết đôi

Danh sách liên kết đôi được khởi tạo với các node chứa xác suất được sắp thứ tự tăng, thứ tự này được bảo đảm suốt tác vụ. Cây được tạo theo phương pháp bottom-up. Hai node đầu danh sách sẽ được loại ra khỏi danh sách để tạo một cây mới, node gốc của cây mới này lại được chèn vào danh sách từ phía cuối lên sao cho thứ tự trong danh sách không bị thay đổi.

b) Dùng đệ quy

Cây Huffman có thể được tạo theo phương pháp top-down bằng đệ quy. Mảng xác suất prob được truyền như tham số cho hàm đệ quy, mảng này sẽ thu nhỏ dần sau mỗi lần gọi đệ quy. Do dùng đệ quy đầu, khi các lần gọi đệ quy thực hiện xong, cây Huffman sẽ được hình thành.

createHuffmanTree(prob)

khai báo p₁, p₂, và cây Huffman Htree;

if chỉ còn hai trị xác suất trong prob

 return một cây với p₁, p₂ là trị trong node lá và p₁ + p₂ là trị trong root;

else

 loại hai trị xác suất nhỏ nhất trong prob, gán chúng cho p₁ và p₂;

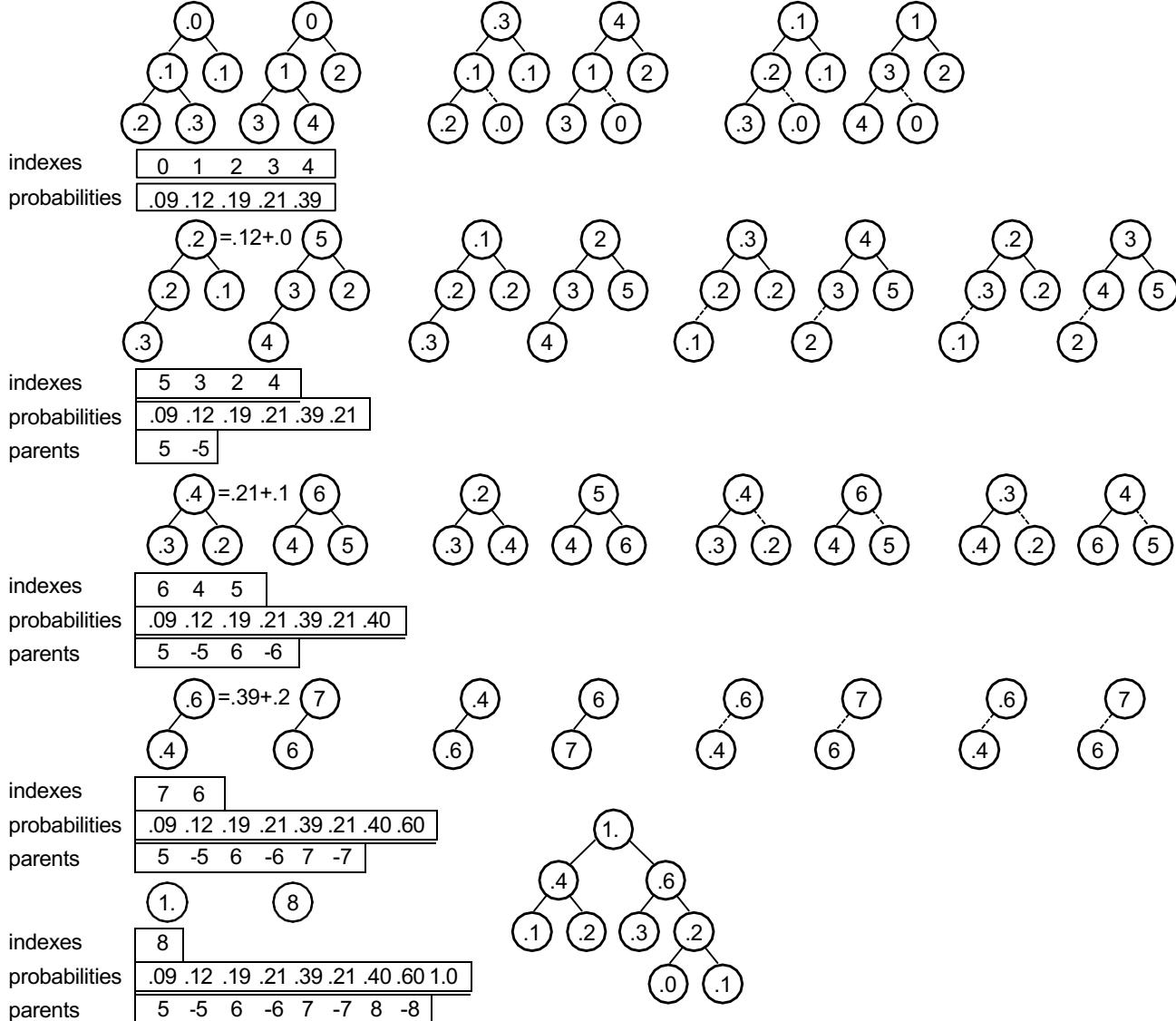
 chèn p₁ + p₂ vào prob;

 Htree = createHuffmanTree(prob);

 trong Htree tạo hai node lá chứa p₁ và p₂ là con của node cha chứa trị (p₁ + p₂);

 return Htree;

c) Dùng minheap



Cây Huffman tạo bằng minheap

Minheap có thuộc tính: node trong bất kỳ chứa trị nhỏ hơn trị của hai con của nó. Node tại đỉnh heap chứa trị nhỏ nhất. Khi loại node khỏi đỉnh heap, node lớn nhất (cuối heap) sẽ được đưa lên đỉnh heap và heap được cấu trúc lại.

Mỗi bước ta cần loại 2 node liên tiếp khỏi đỉnh heap: với node thứ nhất tiến hành như trên, với node thứ hai ta tính tổng xác suất, đưa node tổng này vào đỉnh heap và cấu trúc lại heap. Để có thông tin tạo cây Huffman, ta dùng ba mảng:

- **indexes** lưu chỉ số của các node chứa xác suất còn lại trong heap, chỉ số này tham khảo từ mảng probabilities. Khi mảng này còn một chỉ số (còn một node trong heap), đó là chỉ số của node gốc cây Huffman.

- **probabilities** lưu các xác suất ban đầu và xác suất được tạo mới theo thứ tự tạo. Hai mảng indexes và parents chứa các chỉ số tham khảo đến mảng này.

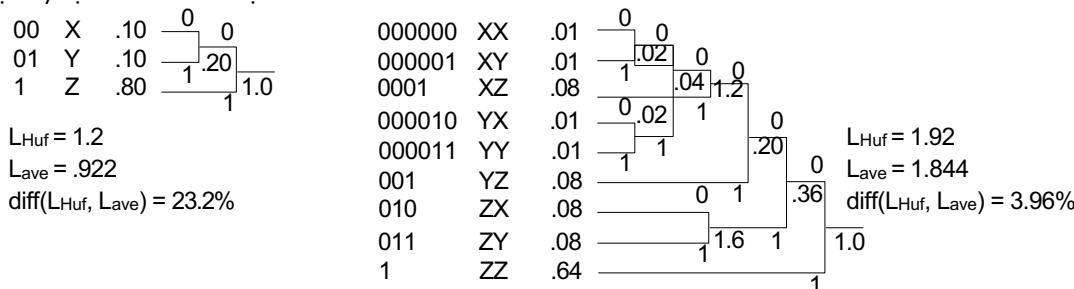
- **parents** được dùng như sau: probabilities[i] là con của probabilities[|parents[i]|]; nếu parents[i] dương nó là con trái, nếu parents[i] âm nó là con phải. Mảng này cũng được dùng khi tạo codeword (tạo cây Huffman rồi xuất codeword).

3. Mã hóa và giải mã bằng giải thuật Huffman

Giả sử ta đã tạo được cây Huffman và có bảng kết quả là: A (11), B (01), C (00), D (101), E (100).

Để mã hóa khi truyền, thay vì gửi các symbol, codeword tương đương với symbol được gửi. Ví dụ, thay vì gửi ABAAD, chuỗi 11011111101 sẽ được chuyển, số bit trên ký tự trung bình là $11/5 = 2.2$, rất gần với Lave 2.09.

Ngoài ra, ta có thể cải thiện hiệu quả nén bằng cách xây dựng cây Huffman dựa trên xác suất xuất hiện *cặp ký tự*, thay vì xác suất xuất hiện ký tự đơn. Xem ví dụ sau:



Cải thiện L_{Huf} bằng cách xây dựng cây Huffman với *cặp ký tự* (phải), thay vì ký tự đơn (trái).

Để giải mã thông điệp, bên nhận thông điệp phải biết bảng chuyển đổi. Bảng chuyển đổi được bên mã hóa gửi kèm thông điệp mã hóa. Dữ liệu thêm này không đáng kể nếu thông điệp được mã hóa có kích thước lớn.

Bên nhận dùng bảng chuyển đổi nhận được, tạo lại cây Huffman giống với cây được tạo khi mã hóa, nhưng có các node lá chứa các symbol thay vì xác suất của nó. Dùng cây này, ta giải mã được các symbol duy nhất, bằng cách từ gốc đi theo các số 0 (sang trái) và 1 (sang phải) cho đến node lá chứa symbol. Ví dụ, khi nhận 1001101. Như vậy, ta đi root - 1 - 0 - 0 đến node lá chứa E; đi root - 1 - 1 đến node lá chứa A; đi root - 0 - 1 đến node lá chứa B. Kết quả giải mã là EAB.

4. Mã hóa Huffman thích ứng (Adaptive Huffman)

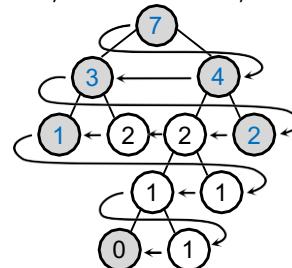
Khi mã hóa Huffman tĩnh, tần suất xuất hiện các symbol phải được thống kê trên dữ liệu đầu vào. Cách này có một số khuyết điểm: dữ liệu đầu vào có kích thước lớn nên quá trình tiền xử lý mất nhiều thời gian, dữ liệu đầu vào không được *nhận biết toàn bộ* vào thời điểm truyền. Trong trường hợp đó, mã hóa Huffman thích ứng là một giải pháp khả thi.

Kỹ thuật mã hóa Huffman thích ứng, còn gọi là mã hóa Huffman động, được phát minh đầu tiên bởi Robert G. Gallager và được cải tiến bởi Donald Knuth. Giải thuật được xây dựng trên *thuộc tính anh em* (sibling property) như sau:

Cho cây Huffman với mỗi node có anh em (node cùng mức) ngoại trừ node root, kết quả việc duyệt theo chiều rộng từ phải-sang-trái là một danh sách node với tần số không tăng. Cây như vậy gọi là cây Huffman có thuộc tính anh em.

Trong mã hóa Huffman thích ứng, cây có một bộ đếm cho mỗi symbol, và bộ đếm được cập nhật mỗi khi symbol đầu vào tương ứng được mã hóa. Mỗi lần cập nhật bộ đếm, cần kiểm tra thuộc tính anh em vẫn còn đảm bảo. Nếu vi phạm thuộc tính anh em, cây cần được tái cấu trúc để khôi phục thuộc tính đó. Dưới đây là cách thực hiện điều này.

Đầu tiên, giả thiết rằng giải thuật duy trì một danh sách liên kết đôi nodes chứa các node của cây theo thứ tự duyệt cây theo chiều rộng từ *phải-sang-trái*. Một block_i là một phần của danh sách sao cho mỗi node trong block_i có chung tần suất là i, node đầu tiên trong mỗi block được gọi là *leader*. Ví dụ, hình dưới là cây Huffman với danh sách nodes theo tần suất = (7 4 3 2 2 2 1 1 1 0) có sáu block: block₇, block₄, block₃, block₂, block₁ và block₀, với leader của mỗi block được in đậm.



Thuộc tính anh em, theo thứ tự từ trên xuống dưới, từ phải sang trái

Khởi đầu, cây có một 0-node chứa tất cả các symbol chưa dùng (còn gọi là NYT, Not Yet Transmitted).

- Nếu symbol nhập vào xuất hiện lần đầu, 0-node sẽ chia làm hai: 0-node mới chứa các symbol còn lại, và node chứa symbol đang xét với bộ đếm được thiết lập bằng 1; cả hai node đều trở thành con của một node cha có bộ đếm cũng thiết lập bằng 1. Sau khi symbol đó được loại khỏi danh sách symbol trong 0-node, vị trí của nó bị chiếm bởi symbol *cuối danh sách*.

- Nếu symbol nhập vào đã có tại node p trong cây thì bộ đếm của node p được tăng. Tuy nhiên, việc tăng này có thể ảnh hưởng đến thuộc tính anh em; nếu vi phạm, thuộc tính anh em được khôi phục bằng cách hoán chuyển node p với leader của block mà p đang nằm trong, ngoại trừ trường hợp leader là cha của p. Leader được tìm bằng cách đi trong nodes từ p đến đầu danh sách. Nếu p thuộc về block_i trước khi tăng, việc hoán chuyển p với leader của block_i sẽ làm p trở thành node cuối của block_{i+1}. Sau đó, bộ đếm của cha mới của p được tăng, điều này có thể dẫn tới việc biến đổi cây để khôi phục thuộc tính anh em. Quá trình này tiếp tục cho đến khi đến root.

Với mỗi symbol nhập vào, ta có hai kiểu lấy codeword:

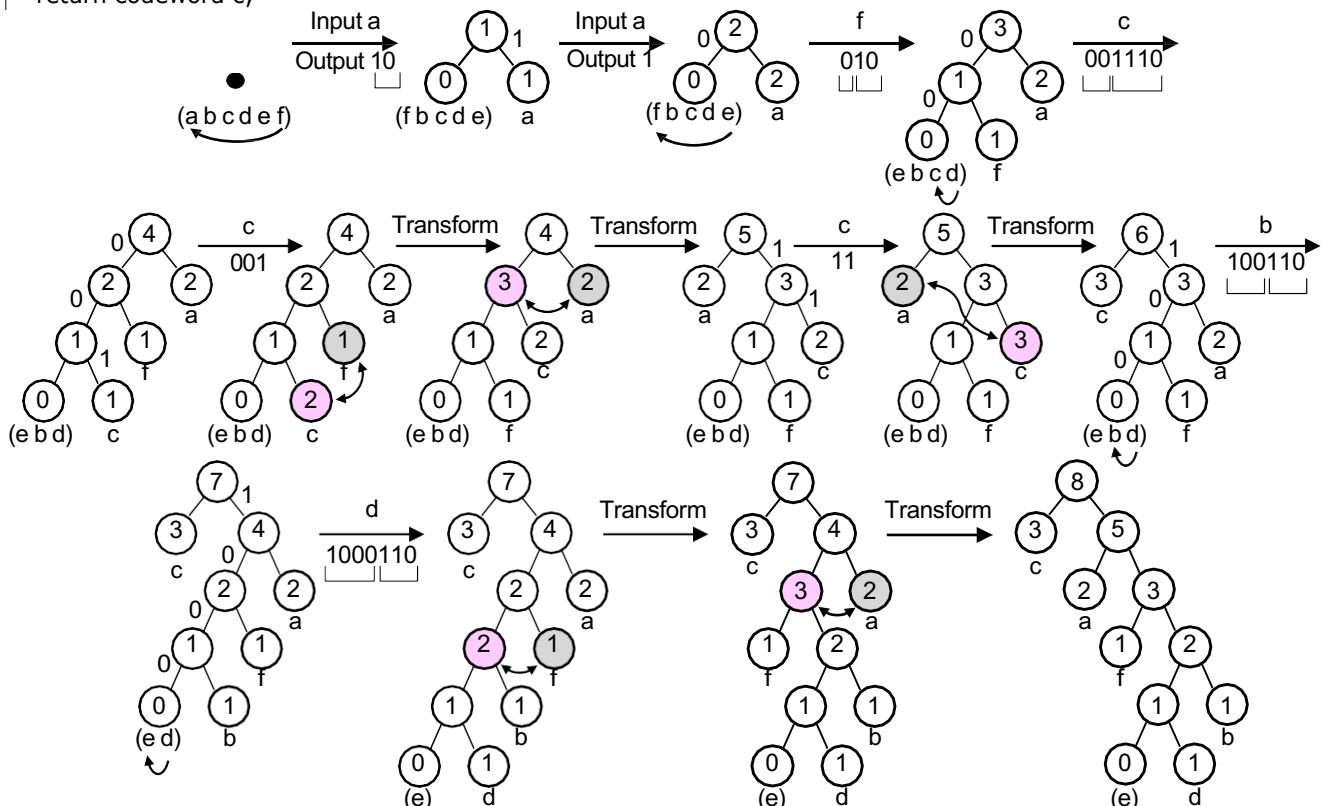
- Nếu symbol nhập vào đã xuất hiện thì thủ tục mã hóa bình thường được áp dụng: quét cây Huffman từ gốc đến node giữ symbol đó *trước bất kỳ chuyển đổi nào* trong cây.

- Nếu symbol nhập vào xuất hiện lần đầu, codeword là kết hợp của: codeword của 0-node hiện tại + số *các số 1* chỉ vị trí của symbol đó trong danh sách các symbol chưa dùng chứa tại 0-node (tính từ trái qua), theo sau là 0 (dùng phân biệt với codeword kế tiếp). Ví dụ, khi symbol c được mã hóa lần đầu, codeword của nó 001110, là sự kết hợp của 00 (codeword của 0-node), 1110 (c chiếm vị trí thứ ba trong danh sách các symbol chưa dùng chứa tại 0-node). Hai phần của một codeword này được đánh dấu tách riêng trong hình dưới.

Giải thuật mã hóa Huffman động:

FGKDYNAMICHUFFMANENCODING(symbol s)

p = node lá chứa symbol s;
 c = Huffman codeword cho s;
 if p là 0-node
 c = c nối với số 1 thể hiện vị trí của s trong 0-node, và nối với 0;
 ghi symbol cuối trong 0-node lên s trong node này;
 tạo node q mới cho symbol s và đặt bộ đếm của nó là 1;
 p = node mới trở thành node cha của 0-node và node q;
 bộ đếm của p là 1;
 đưa hai node mới vào nodes;
 else
 tăng bộ đếm của p;
 while p không là root
 if p vi phạm thuộc tính anh em
 if leader của block_i chứa p không là parent(p)
 hoán chuyển p với leader;
 p = parent(p);
 tăng bộ đếm của p;
 return codeword c;



Truyền thông điệp "aafcccbd" bằng cách dùng giải thuật Huffman động
 Node xám: leader của block chứa node vi phạm thuộc tính anh em

- Khởi đầu, cây chỉ chứa 0-node với tất cả các ký tự nguồn (a, b, c, d, e, f). Sau ký tự nhập đầu tiên, a, chỉ có codeword cho vị trí chiếm bởi a trong 0-node được xuất. Bởi vì a có vị trí đầu tiên trong 0-node, số 1 được xuất, theo sau đó là 0. Ký tự cuối cùng trong 0-node (f) được đặt lên vị trí của a, và một node riêng được tạo ra cho ký tự a có bộ đếm là 1. Một node mới thành cha của 0-node và node chứa a.
- Ký tự nhập thứ hai vẫn là a, 1 được xuất, chính là Huffman codeword của lá chứa a. Tần suất của a tăng lên 2, điều này vi phạm thuộc tính anh em, nhưng do leader của block là cha của node p (tức node a), không có phép hoán đổi nào được thực hiện; chỉ có p được cập nhật để chỉ đến cha nó và tần suất của p được tăng.
- Ký tự nhập thứ ba là f, là ký tự xuất hiện lần đầu, codeword của f (010) gồm: Huffman codeword cho 0-node (0) + 1 số 1 tương ứng với vị trí của f trong 0-node, theo sau là 0 (10). Ký tự e được đưa lên vị trí của f trong 0-node, một node lá mới cho f được tạo và một node mới trở thành cha của 0-node và node chứa f. Node p là cha của lá f không vi phạm thuộc tính anh em nên p được cập nhật, p = parent(p), do đó trở thành root và bộ đếm của nó được tăng lên.
- Ký tự nhập thứ tư là c, xuất hiện lần đầu, codeword của c (100110) gồm Huffman codeword cho 0-node (100) + ba số 1 do c là ký tự thứ 3 trong 0-node, theo sau là 0 (1110). Sau đó, d được đưa lên vị trí của c trong 0-node và c được đưa vào một node lá mới được tạo, p được cập nhật hai lần, cho phép tăng bộ đếm của hai node: con trái của root và root.
- Ký tự nhập tiếp theo là c; vì thế, đầu tiên Huffman codeword cho node chứa nó được lấy là 001. Tiếp theo, bởi vì thuộc tính anh em bị vi phạm nên node p (tức node c), được hoán chuyển với leader của block₁ là f. Sau đó, p = parent(p) và cha mới p của node c được tăng bộ đếm, dẫn đến vi phạm tiếp thuộc tính anh em, phải hoán chuyển node p với leader của block₂ là a. Tiếp theo, p = parent(p), bộ đếm node p được tăng, nhưng do nó là root nên quá trình cập nhật cây hoàn thành.

6. Ký tự nhập thứ sáu vẫn là c, có trong cây, do đó đầu tiên Huffman codeword của node c là 11 được gửi và bộ đếm của node c được tăng lên. Node p (tức node c) vi phạm thuộc tính anh em nên p được hoán chuyển với leader block₃ là a. Bây giờ, p = parent(p), bộ đếm của p được tăng lên và bởi vì p là root, việc biến đổi cây được kết thúc cho ký tự này. Các bước còn lại có thể theo dõi trên hình.

III. Mã hóa chiều dài dòng chạy (Run-Length Encoding)

Một dòng chạy (run) được định nghĩa là một chuỗi các ký tự giống nhau. Ví dụ, chuỗi s = "aaabba" có 3 run: run "a", run "b" và run "a". Phương pháp mã hóa run-length là một phương pháp lợi dụng sự hiện diện của các run và thể hiện chúng dưới dạng tóm tắt hơn, nhỏ gọn hơn.

Nếu các run chứa các ký tự giống nhau, như trong chuỗi s = "nnnn***r%%%%%%%", thay vì truyền chuỗi này, ta truyền thông tin của run. Mỗi run được mã hóa bằng một cặp (n, ch), với ch là một ký tự và n là một số nguyên đại diện cho số ký tự ch có trong run. Chuỗi s được mã hóa dưới dạng 4n3*1r7%. Tuy vậy, một vấn đề xảy ra khi ký tự ch là một chữ số, như trong trường hợp 1111111111544444, run sẽ được biểu diễn là 1111554 (11 chữ số 1, 1 chữ số 5 và 5 chữ số 4). Chính vì vậy, đối với mỗi run, thay vì sử dụng số n, ta có thể dùng một ký tự có mã ASCII là n để thay thế. Ví dụ, run chứa 43 ký tự "c" liên tục có thể được biểu diễn là +c ("+" có mã ASCII là 43), và run chứa 49 số 1 có thể biểu diễn là 11 ("1" có mã ASCII là 49).

Phương pháp này chỉ hiệu quả khi một run có ít nhất 2 ký tự, bởi vì đối với run chỉ có một ký tự, codeword trả về sẽ là 100% ch. Điều này dẫn đến nhu cầu sử dụng một dấu hiệu (marker) để chỉ ra rằng chuỗi đang được truyền đi là ở dạng nén hay là dạng không nén. Như vậy, cần có bộ 3 ký tự (cm, ch, n) để thể hiện một run: dấu hiệu nén - cm (compression marker), ký tự - ch (literal character) và bộ đếm - n (counter).

Việc chọn dấu hiệu nén phải rất khéo léo để tránh gây nhầm lẫn với ký tự đang được truyền. Nếu một tập tin văn bản được truyền đi, ký tự '~'+1 có thể được lựa chọn, do ký tự ~ ít xuất hiện trong văn bản. Trong trường hợp dấu hiệu nén xuất hiện trong dữ liệu đầu vào, ta giải quyết bằng cách truyền dấu hiệu nén 2 lần. Bộ giải mã sẽ bỏ đi 1 trong 2 dấu hiệu nén khi nhận được chúng cùng lúc và giữ cái còn lại như là một phần của dữ liệu đang được nhận, điều này giống như xử lý chuỗi escape trong C. Ngoài ra, các run chứa dấu hiệu nén không được gửi dưới dạng nén.

Việc nén các run trả về kết quả là một chuỗi 3 ký tự. Vì thế, phương pháp này nên được áp dụng vào các run có ít nhất 4 ký tự. Chiều dài lớn nhất của một run có thể biểu diễn được bằng bộ ba (cm, ch, n) là 255 với ký tự n là 8-bit ASCII. Nhưng do chỉ có những run chứa 4 ký tự được mã hóa, n biểu diễn cho số ký tự có thật trong run trừ đi 4; ví dụ, nếu n = 1, thì có 5 ký tự trong run. Trong trường hợp này, chiều dài run dài nhất có thể biểu diễn thông qua một bộ ba là 259 ký tự.

Mã hóa run-length chỉ có hiệu quả nhất định đối với các tập tin văn bản mà trong đó, các ký tự blank (khoảng trắng) có xu hướng lặp lại. Trong trường hợp này, văn bản được xử lý trước khi nén bằng *null suppression*, ta nén chỉ các run chứa blank và không cần xác định ký tự nào đang được nén. Kết quả là cặp (cm, n) được dùng cho các run có 3 hoặc nhiều blank.

Phương pháp mã hóa run-length rất hữu dụng khi áp dụng vào các tập tin văn bản, bản ghi trong cơ sở dữ liệu, hình ảnh.

Một khuyết điểm lớn của phương pháp này là nó hoàn toàn phụ thuộc vào số run xuất hiện. Cụ thể, phương pháp này không thể tự nó nhận ra tần số xuất hiện của các ký tự nhất định. Ví dụ, AAAABBBB có thể được nén, vì nó được có hai run, nhưng ABABABAB lại không thể được nén mặc dù cả 2 thông điệp được tạo ra từ những ký tự giống nhau. Mặt khác, ABABABAB được nén với phương pháp mã hóa Huffman cũng cho ra số codeword tương đương AAAABBBB mà không cần quan tâm sự có mặt của các run.

IV. Mã Ziv-Lempel

Các giải thuật nén đã thảo luận phải biết trước thông tin về dữ liệu trước khi tiến hành mã hóa. Ví dụ mã hóa Huffman tinh cần biết trước tần suất xuất hiện các symbol, trước khi codeword được gán cho các symbol. Mã hóa Huffman động có thể khắc phục điều này, không dựa trên thông tin biết trước về nguồn dữ liệu đầu, mà thu thập thông tin đó trong quá trình mã hóa dữ liệu. Phương pháp như vậy gọi là mã hóa phổ dụng (universal encoding scheme), mã Zip-Lempel là một ví dụ điển hình.

1. LZ77

Xem trực quan giải thuật LZ77 tại: <http://projects.hudeco.net/diplomovka/online/ucebnica/applets/AppletLZ77.html>

a) Nén LZ77

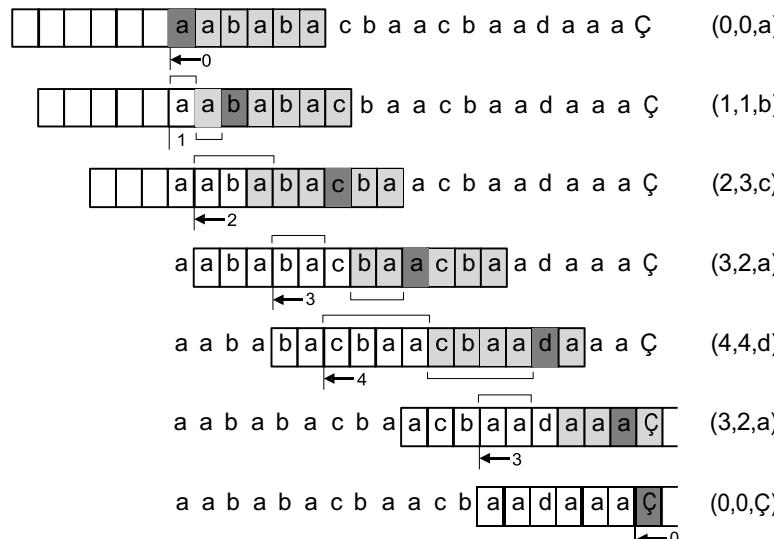
Ý tưởng chính của giải thuật này là dùng một phần dữ liệu đã xét như một tự điển. Giải thuật cần một cửa sổ đệm, trượt từ trái sang phải. Cửa sổ đệm có hai phần: [vùng đệm tìm kiếm][vùng đệm đọc]

- Vùng đệm tìm kiếm: chứa các ký tự vừa được đọc, dùng làm tự điển cho các ký tự chờ nén.
- Vùng đệm đọc: chứa các ký tự chờ nén. Chuỗi bắt đầu ngay từ đầu vùng đệm này sẽ được tìm kiếm trong tự điển. Nếu so trùng, chuỗi này (và ký tự ngay sau nó) sẽ được nén bằng cách thay thế bởi bộ ba (offset, length, next).

Kích thước các vùng đệm rõ ràng ảnh hưởng đến hiệu quả nén. Nếu kích thước vùng đệm tìm kiếm lớn, tự điển sẽ lớn và khả năng nén có thể tăng nhưng tốc độ nén giảm do tăng thời gian tìm kiếm.

Giải thuật nén LZ77 như sau:

- | |
|---|
| <ul style="list-style-type: none"> - Đặt con trỏ mã hóa ngay đầu vùng đệm đọc. - Lắp trong khi vùng đệm đọc còn chưa rỗng. - Tìm chuỗi dài nhất của vùng đệm đọc (tính từ đầu vùng đệm đọc, tức con trỏ mã hóa) có trong vùng đệm tìm kiếm. - Xuất bộ ba (offset, length, next). Trong đó: <ul style="list-style-type: none"> offset: vị trí tìm thấy chuỗi so trùng trong vùng đệm tìm kiếm, offset tính từ phải sang trái. length: chiều dài của chuỗi so trùng. next: ký tự sai khác đầu tiên sau chuỗi so trùng trong vùng đệm đọc. - Trượt cửa sổ đệm sang phải (length + 1) ký tự. |
|---|



Nén LZ77 chuỗi: "aababachacaacbaadada", xuất các bộ ba. Màu trắng: vùng đệm tìm kiếm.

Màu xám: vùng đệm đọc; xám đậm: ký tự next. Số có mũi tên: offset (tính từ phải sang trái).

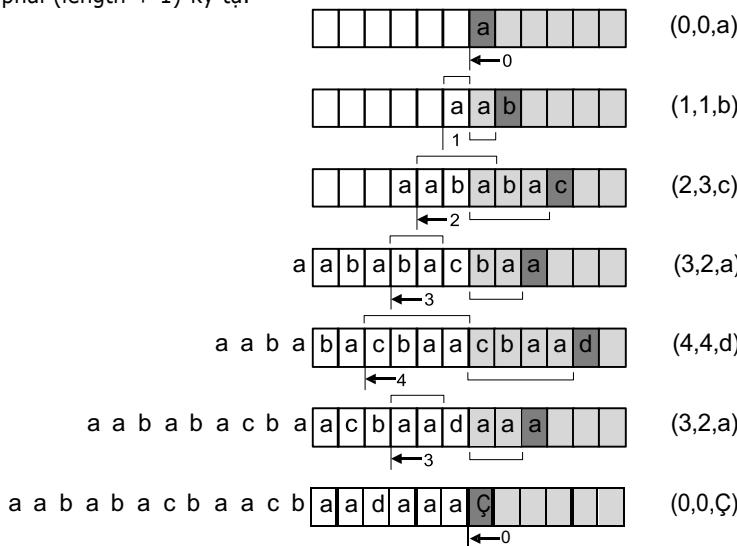
Khung phía dưới: chuỗi ký tự được đọc để so trùng. Khung phía trên: chuỗi ký tự so trùng, chiều dài length.

Chú ý đến trường hợp xuất (2,3,c). Chuỗi aba trong vùng đệm đọc được so trùng với chuỗi aba bắt đầu từ offset 2 (tính từ phải sang trái) của vùng đệm tìm kiếm. Điều này cho thấy offset bắt đầu từ 0 (và tăng lên) trong vùng đệm tìm kiếm, chuỗi so trùng không nhất thiết phải nằm trọn trong vùng đệm tìm kiếm.

b) Giải nén LZ77

Giải thuật giải nén LZ77 như sau:

- Đọc từng bộ ba (offset, length, next).
- Từ vị trí offset trong vùng đệm tìm kiếm, đọc từ có chiều dài length.
- Ghi từ tìm được vào vùng đệm đọc.
- Ghi next tiếp theo sau.
- Trượt cửa sổ đệm sang phải (length + 1) ký tự.



Giải nén LZ77 chuỗi: (0,0,a)(1,1,b)(2,3,c)(3,2,a)(4,4,d)(3,2,a)(0,0,\$). Màu trắng: vùng đệm tìm kiếm.

Màu xám: vùng đệm đọc; xám đậm: ký tự next. Số có mũi tên: offset (tính từ phải sang trái).

Khung phía trên: chuỗi ký tự được đọc với chiều dài length. Khung phía dưới: chuỗi ký tự được ghi vào.

Chú ý đến trường hợp giải nén (2,3,c). Tại offset 2 (tính từ phải sang trái) của vùng đệm tìm kiếm, đọc 3 (length) ký tự và ghi vào đầu vùng đệm đọc. Vì đọc từng ký tự nên ta đọc được ký tự thứ ba, a, vừa mới ghi vào trước đó.

2. LZW

Một phiên bản thường được sử dụng của giải thuật Ziv-Lempel là LZW, LZW dùng một tự điển với chỉ số là codeword, được tạo trong quá trình truyền dữ liệu.

Xem trực quan giải thuật LZ77 tại: <http://projects.hudecof.net/diplomovka/online/ucebnica/applets/AppletLZW.html>

a) Nén LZW

Giải thuật nén LZW như sau:

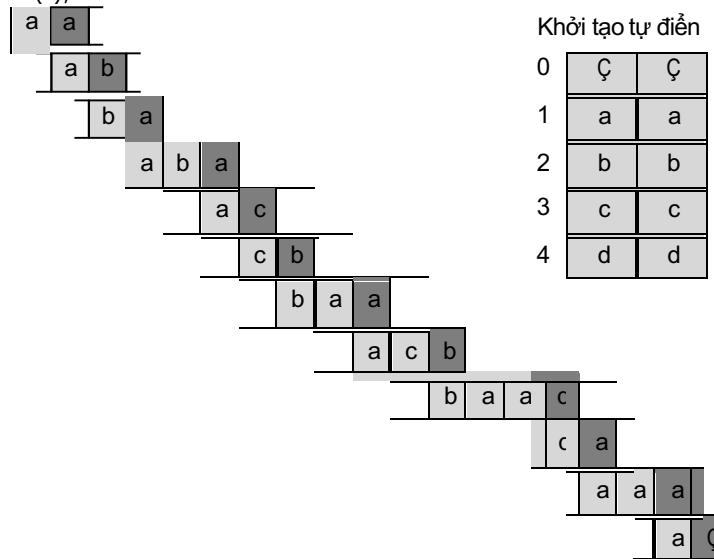
LZWcompress()

- nhập tất cả các ký tự vào tự điển;
- s = ký tự đầu tiên từ ngõ nhập;
- while còn ký tự // hoặc gặp ký tự kết thúc, ví dụ \$

```

read ký tự c;
if s+c có trong tự điển
  s = s+c;
else          // s+c không có trong tự điển
  xuất codeword(s);
  nhập s+c vào tự điển;
  s = c;
xuất các codeword(s);

```



Nén LZW chuỗi: "aababacbaacbaad", xuất "1 1 2 6 1 3 7 9 11 4 5 2"

Trái: xử lý chuỗi, màu xám nhạt là s (sẽ xuất), xám đậm là c, s+c sẽ đưa vào tự điển

Phải: tự điển, bao gồm [codeword:chuỗi đầy đủ:chuỗi viết tắt]

Chuỗi s luôn có độ dài ít nhất là một ký tự. Sau khi đọc vào một ký tự mới, chuỗi ghép s+c được kiểm tra trong tự điển:

- Nếu s+c có trong tự điển, một ký tự mới sẽ được ghép thêm vào s+c và kiểm tra tiếp.

- Nếu s+c không có trong tự điển, codeword cho s được xuất; s+c được lưu vào tự điển và s được khởi tạo lại với c.

Một yếu tố quan trọng là tổ chức của tự điển. Số mục của tự điển có thể lên đến hàng nghìn, vì thế cần dùng một phương thức tìm kiếm hiệu quả. Một vấn đề khác là kích thước chuỗi lưu vào tự điển, kích thước của chúng có khuynh hướng tăng dần. Vấn đề này được giải quyết bằng cách lưu vào tự điển chuỗi viết tắt bao gồm: tiền tố và ký tự cuối cùng của chuỗi. Ví dụ, nếu "ba" được gán codeword là 7, thì "baa" được lưu trong bảng như sau: codeword cho tiền tố "ba" của nó là 7, và ký tự cuối cùng của nó là "a"; ta có được 7a. Bằng cách này, tất cả dữ liệu vào của tự điển sẽ có cùng độ dài. Dẫn đến vấn đề tìm kiếm được giải quyết tốt bằng các hàm băm.

b) Giải nén LZW

Để giải mã, một tự điển tương tự được tạo bằng cách cập nhật nó với từng codeword đi vào ngoại trừ codeword đầu tiên. Chuỗi tạo từ codeword trước đó (priorcodeword) và codeword đi vào, sẽ được cập nhật vào tự điển tùy theo codeword đi vào có tìm thấy trong tự điển hay không.

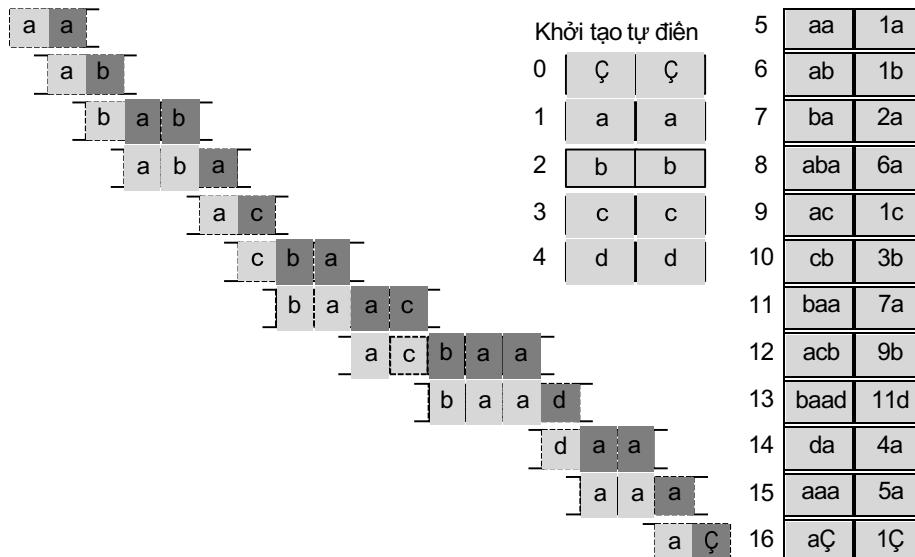
Giải thuật giải nén LZW như sau:

LZWdecompress()

```

nhập tất cả các ký tự vào tự điển;
đọc codeword đầu tiên vào priorcodeword và xuất ký tự tương ứng với nó;
while còn codeword // hoặc gấp codeword kết thúc, ví dụ 0
  đọc codeword;
  if codeword không có trong tự điển
    nhập vào tự điển string(priorcodeword) + ký tự đầu(string(priorcodeword));
    xuất string(priorcodeword) + ký tự đầu(string(priorcodeword));
  else
    nhập vào tự điển string(priorcodeword) + ký tự đầu(string(codeword));
    xuất string(codeword);
    priorcodeword = codeword;

```



Giải nén LZW chuỗi: "1 1 2 6 1 3 7 9 11 4 5 2", xuất "aababacbaacbaadaaa". Trái: xử lý chuỗi; phải: tự điển
Xám nhạt: ký tự tương ứng với priorcodeword, tra trong tự điển

Xám đậm: ký tự tương ứng với codeword, tra trong tự điển

Khung có nét đứt: codeword mới sẽ được cập nhật vào tự điển

Ví dụ trên không có trường hợp rơi vào phần if trong giải thuật. Đó là khi codeword đi vào không tìm thấy trong tự điển. Vấn đề này xảy ra khi một chuỗi đang được giải mã có chứa một chuỗi con "cScScS", "c" là một ký tự và "cS" đã tồn tại trong tự điển. Khi đó, codeword mới đưa vào tự điển chỉ được tạo dựa trên priorcodeword.

Tài liệu tham khảo

(Theo năm xuất bản)

- [1] Adam Drozdek - **Data Structures and Algorithms in Java, Fourth Edition** - Cengage Learning Asia, 2013. ISBN: 9-814-39278-2
- [2] Robert Sedgewick, Philippe Flajolet - **An Introduction to the Analysis of Algorithms, Second Edition** - Addison Wesley, 2013. ISBN: 0-321-90575-X
- [3] Robert Sedgewick, Kevin Wayne - **Algorithms, 4th Edition** - Addison Wesley, 2011. ISBN: 0-321-57351-X
- [4] Clifford A. Shaffer - **Data Structures and Algorithm Analysis in Java, Third Edition** - Dovers Publications, 2011. ISBN: 0-486-48581-1
- [5] William J. Collins - **Data structures and the Java collections framework, Third Edition** - John Wiley & Sons, Inc., 2011. ISBN 978-0-470-48267-4
- [6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein - **Introduction to Algorithms, Third Edition** - Massachusetts Institute of Technology, 2009. ISBN 978-0-262-53305-8